Este exemplar corresponde à redação final da Tese / Dissertação devidamente corrigida e defendida por: Flanklin Robert Araujo França

e aprovada pela **Banca Examinadora.**

Campinas, 29 de julho de 1999

COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Análise e medidas de desempenho dos protocolos de comunicação do Plan 9 e uma comparação com o sistema UNIX

Franklin Robert Araujo França

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Análise e medidas de desempenho dos protocolos de comunicação do Plan 9 e uma comparação com o sistema UNIX

Franklin Robert Araujo França

Julho de 1999

Banca Examinadora:

- Célio Cardoso Guimaraes (Orientador)
- Maurício Ferreira Magalhães
  Faculdade de Engenharia Elétrica e Computação - UNICAMP
- Ricardo de Oliveira Anido
  Instituto de Computação - UNICAMP
- Paulo Lício de Geus (suplente)
  Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

França, Franklin Robert Araujo

F844a Análise e medidas de desempenho dos protocolos de comunicação do Plan 9 e uma comparação com o sistema UNIX / Franklin Robert Araujo França -- Campinas, [S.P. :s.n.], 1999.

Orientador : Célio Cardoso Guimarães

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas operacionais (Computadores). 2. UNIX (Sistema operacional de computador). 3. LINUX (Sistema operacional de computador). I. Guimarães, Célio Cardoso. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Análise e medidas de desempenho dos protocolos de comunicação do Plan 9 e uma comparação com o sistema UNIX

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Franklin Robert Araujo França e aprovada pela Banca Examinadora.

Campinas, 2 de Julho de 1999.

Célio Cardoso Guimaraes (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Em tudo quanto for fazer, lembre-se de colocar Deus em primeiro lugar. Ele guiará os seus passos e você andará pelo caminho do sucesso.

(Provérbios 3:6 - A Bíblia Viva)

Ao grande mestre Jesus Cristo.

Ao meu querido pai.

Agradecimentos

Obrigado Senhor Deus por estar ensinando-me a andar conforme a tua palavra. Obrigado pelos novos amigos e por meus estudos.

"Jesus olhou para trás [e] viu que eles o acompanhavam..." André e seu amigo tentavam entrar em contato com Ele. Então Jesus os convidou: "Venham e vejam." Assim, os dois "foram e viram".

No dia seguinte, André foi depressa buscar Simão Pedro e disse-lhe: "Achamos o Messias."

Como André, meu amigo André deseja que outros encontrem o Salvador. E foi assim comigo. Convidou-me para morar com ele, já pensando em falar de Jesus para mim. Obrigado meu irmão. Hoje Jesus está na minha vida, graças ao seu empenho e ao amor de Deus. Obrigado também por toda ajuda que tu me deste. Tu és realmente uma benção em minha vida.

Seguir os passos de Jesus significa viver uma vida de amor, como Ele viveu. Obrigado meus amigos da Igreja Batista Nova Aliança e em especial ao grupo de jovens, pela convivência no amor de Jesus durante esses dois anos. Vocês são a minha família aqui em Campinas. Obrigado Xavier, Daniel, Calvin, Alex e Ebenézer pelos conselhos e ensinamentos bíblicos.

Ao meu orientador, Prof. Célio Guimarães, por toda ajuda, apoio e incentivo constante no desenvolvimento desta dissertação. Obrigado também ao Celinho e a Tânia pela amizade.

Aos professores do Departamento de Ciências da Computação da Universidade Federal do Maranhão e aos professores do Instituto de Computação da UNICAMP.

Ao Prof. Nelson Machado por ter cedido sua máquina para a instalação do Plan 9. Sem ela não teríamos o servidor de CPU. A Camila pela ajuda na parte de UNIX e ao Bodê na parte de hardware.

Ao meu pai, Francisco Silva França, pelo apoio, dedicação e grande amor que sempre teve por mim. A minha mãe, madrasta, irmão, irmãs, minha sobrinha Amanda e ao Mateus.

Aos amigos da UNICAMP, especialmente: Arlindo, Erlon, Cláudio, Guilherme Júnior, Guilherme Pimentel, César, Emerson e Hélio pela companhia e grande amizade.

Aos amigos da República I: Alexandre Torrezan, Rogério, Ivan, Alexandre Tobias, Miller e Euclides. E da República II: Luís, Daniel Baiano, Daniel Peruano, Waldson, Rafael, Gustavo, Marcelo, Bruno, Carlos e Urtiga. Obrigado pela convivência e amizade.

Aos meus amigos Guilherme Júnior e Emerson pela ajuda na prévia da apresentação.

Ao meu irmão Jefferson por ter vindo de São Luís - Maranhão para assistir à minha apresentação.

Aos Professores Maurício Magalhães e Ricardo Anido pelas sugestões de melhoria deste trabalho.

A CAPES, CNPq e FAPEMA pelo apoio financeiro.

Resumo

Plan 9 é um sistema operacional distribuído desenvolvido no Computing Sciences Research Center da Bell Laboratories. Plan 9 suporta várias arquiteturas de hardware e usa três tipos de componentes: terminais, servidores de arquivos e servidores de CPU. No Plan 9 todos os objetos do sistema apresentam-se como arquivos que podem existir tanto local quanto remotamente e respondem a um protocolo chamado 9P, que executa sobre o protocolo de transporte IL (Internet Link) e que têm funções semelhantes ao NFS.

Nesta dissertação é feito um estudo do sistema Plan 9 concentrando-se nos recursos de comunicação suportados pelo kernel entre processos locais e remotos. É feita uma comparação do desempenho desses mecanismos de comunicação através de micro-benchmarks e de macro-benchmarks, nos sistemas operacionais Plan 9, Linux e UNIX/SunOS. O desempenho da comunicação remota do Plan 9 é também comparado com UNIX/NFS.

Palavras chaves: análise de desempenho, micro-benchmarks, macro-benchmarks, sistemas operacionais distribuídos, Plan 9, Linux, UNIX, SunOS.

Abstract

Plan 9 is a distributed operating system developed at the Bell Labs' Computing Sciences Research Center. Plan 9 supports several hardware architectures and uses three types of components: terminals, file servers and CPU servers. All Plan 9 resources look like files which can be local or remote, and respond to a protocol named 9P which runs over the transport protocol IL (Internet Link) and which are similar to NFS.

In this dissertation we study the Plan 9 system, focusing on the kernel communication facilities between local and remote processes. The performance of these communication mechanisms are compared through micro and macro-benchmarks in the Plan 9, Linux and UNIX/SunOS operating systems. The performance of Plan 9 remote communication is also compared with UNIX/NFS.

Palavras chaves: performance analysis, micro-benchmarks, macro-benchmarks, distributed operating systems, Plan 9, Linux, UNIX, SunOS.

Conteúdo

Agradecimentos vii

Resumo ix

Abstract x

1 Introdução 1
- 1.1 Objetivos da dissertação . . . 1
- 1.2 Organização da dissertação . . . 2
- 1.3 Sistemas Operacionais relacionados . . . 3
  - 1.3.1 Amoeba . . . 3
  - 1.3.2 Mach . . . 5

2 Uma visão geral do sistema Plan 9 10
- 2.1 Decomposição da estrutura funcional . . . 11
- 2.2 A organização de redes no Plan 9 . . . 13
  - 2.2.1 Modelo de Processos e Programação Paralela . . . 13
- 2.3 Portabilidade . . . 16
- 2.4 Protocolos de Comunicação . . . 17
  - 2.4.1 O protocolo IL . . . 18
- 2.5 Autenticação . . . 18
  - 2.5.1 Autenticando conexões externas . . . 20
  - 2.5.2 Usuários especiais . . . 20
  - 2.5.3 Autenticação proxy para serviço de cpu . . . 21
  - 2.5.4 Permissões de arquivos . . . 21
- 2.6 Ferramentas . . . 22
  - 2.6.1 Alef . . . 22
  - 2.6.2 Acme . . . 23
  - 2.6.3 Acid . . . 23
  - 2.6.4 O editor de textos Sam . . . 24

2.6.5 O sistema de janelas $8\frac{1}{2}$ 24

3 Arquitetura do Sistema Plan 9 **25**
3.1 O Protocolo 9P 26
3.2 Espaço de nomes 27
3.3 Servidores básicos do Plan 9 30
3.3.1 Servidor do kernel 32
3.4 Exemplo de serviço: "sistema de arquivo de processos" 34
3.5 Suporte do kernel para redes 36
3.5.1 Dispositivos de protocolo 37
3.5.2 "Banco de dados da rede" 39
3.5.3 Servidor de conexão 41
3.5.4 Rotinas de biblioteca 42
3.5.5 Comunicação entre máquinas Plan 9 e com outros sistemas 44
3.6 O protocolo IL 46
3.6.1 Estabelecimento e término de uma conexão IL 47
3.6.2 Diagrama de Transição de Estados 48
3.6.3 Estrutura do cabeçalho 50
3.6.4 Temporizadores 51

4 Instalação do Plan 9 em ambiente PC **53**
4.1 Distribuição 53
4.2 Instalação 54
4.2.1 Configuração da rede 57
4.2.2 Instalação de um servidor de CPU 59
4.2.3 Administração 60
4.2.4 Kernel 61
4.2.5 Documentação 62
4.3 SPARC 62
4.4 Problemas encontrados 63

5 Medidas de desempenho **64**
5.1 Introdução a benchmarks de sistemas operacionais 64
5.2 Benchmarks usados 65
5.3 Descrição dos benchmarks 67
5.3.1 Micro-benchmarks 67
5.3.2 MAB 69
5.4 Ambiente de execução dos benchmarks 71
5.4.1 MAB 72

5.5 Comentários sobre os resultados dos testes 72
5.5.1 Testes de memória 73
5.5.2 Teste de leitura do buffer-cache 78
5.5.3 Transferência de dados via pipe 79
5.5.4 Transferência de dados via TCP 80
5.5.5 Operações sobre metadados de arquivos 82
5.5.6 Latência de mudança de contexto 83
5.5.7 Outros testes de latência 85
5.5.8 E/S em disco 86
5.5.9 MAB 90

6 Conclusão **92**
6.1 Dificuldades encontradas 92
6.2 Contribuições deste trabalho 92
6.3 Trabalhos futuros 93
6.4 Comentários finais 93

A Terminologia e conceitos básicos **94**
A.1 Introdução 94
A.2 Conceitos básicos 94
A.3 O sistema de arquivos UNIX 98
A.3.1 Sistema de nomes hierárquico 98

Bibliografia **100**

Lista de Tabelas

3.1 Descrição do Protocolo 9P 27

5.1 Medidas de latência local/remota (microssegundos) 85
5.2 Escrita LMDD local (MB/s) 87
5.3 Escrita LMDD remoto (MB/s) 88
5.4 MAB local (s) 90
5.5 MAB remoto 91

Lista de Figuras

1.1 Arquitetura do sistema Amoeba 4
1.2 (a)Threads no Amoeba e (b)Chamada de Procedimento Remoto 5
1.3 Uma capability do Amoeba 5
1.4 Estrutura do sistema Mach 3 6
1.5 Mach 7
1.6 Abstrações básicas do sistema Mach 9

2.1 Estrutura de uma grande distribuição Plan 9. 12

3.1 Espaço de nomes do Plan 9 25
3.2 Visão cliente/servidor antes do mount 28
3.3 Visão do cliente após o mount 28
3.4 Visão dos arquivos após a operação bind 29
3.5 Alterando o espaço de nomes com bind 29
3.6 União dos conteúdos dos diretórios A/B e C/D 30
3.7 Mount device 33
3.8 Servidor no nível do usuário 36
3.9 Árvore de diretórios dos dispositivos de protocolo 38
3.10 Servidor de conexão 41
3.11 IL two-way handshake 48
3.12 Transições de estado do IL 49

5.1 Árvore de diretórios do MAB 70
5.2 bw_mem_rd: mede a taxa de leitura da memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor. 74
5.3 bw_mem_wr: mede a taxa de escrita na memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor. 75
5.4 bw_bzero: mede a taxa de escrita na memória usando a função de sistema *bzero*() ou *memset*(). 76

5.5 bw_mem_cp: taxa obtida ao copiar dados entre dois buffers de memória, alinhados (aligned) ou não em fronteira de página (unaligned), através da libc, em unidades de *bufsize* Kbytes. 77
5.6 bw_file_rd: mede a taxa de releitura de um arquivo de 8MB já cacheado em memória (no "buffer cache"), em unidades do parâmetro *bufsize* Kbytes. . 78
5.7 bw_pipe: mede a taxa obtida ao transferir dados através de um pipe entre dois processos em unidades de *bufsize* Kbytes. 79
5.8 bw_tcp_il: mede a taxa obtida ao transferir dados em unidades de *bufsize* Kbytes entre dois processos conectados via TCP/IL. 81
5.9 lat_fs: mede a latência de operações de metadados sobre arquivos em Kbytes(0 a 10KB). 82
5.10 lat_ctx2: mede a latência média do chaveamento entre 2 processos ao passar um token através de uma seqüência de pipes conectando n processos (n varia de 2 a 20). 84
5.11 Leitura LMDD local (MB/s):mede taxas de transferência para copiar seqüencialmente um arquivo local de tamanho 100MB, em unidades de *bufsize* Kbytes. 87
5.12 lmdd:U9FSxNFS: mede taxas de transferência para copiar seqüencialmente um arquivo remoto de tamanho 8MB, em unidades de *bufsize* Kbytes. . . . 89

Capítulo 1

Introdução

Vários sistemas operacionais distribuídos têm sido projetados. Dos sistemas que foram construídos, poucos saíram do estágio de testes de aplicações distribuídas para tornarem-se um sistema operacional distribuído de uso geral. Isto se deve geralmente ao fato desses sistemas proverem um alto grau de tolerância a falhas, ou por que eles foram construídos no topo de outro sistema operacional, como UNIX, que torna mais fácil o desenvolvimento.

Como uma tentativa em ter um sistema que fosse centralmente administrado e de custo razoável usando microcomputadores e estações de trabalho modernos e baratos como seus elementos de computação, foi desenvolvido no Computing Sciences Research Center da Bell Laboratories pelo mesmo grupo que desenvolveu o UNIX, um sistema operacional distribuído completo incluindo utilitários associados, denominado Plan 9.

Plan 9 tem algumas características interessantes, tais como:

- Distribuição de componentes computacionais e serviços
- Administração centralizada
- Espaço de nomes flexível e personalizado por usuário
- Compilador C independente de arquitetura

O sistema Linux[Cornes97] foi escolhido para comparação com o Plan 9, por executar na mesma arquitetura PC da nossa instalação do Plan 9, e por ter uma excelente implementação nessa arquitetura.

1.1 Objetivos da dissertação

Pelo fato de Plan 9 ser um sistema genuinamente distribuído, uma aplicação típica requer forte interação entre vários componentes remotos do sistema. Por esta razão, a eficiência

dos protocolos de comunicação é da maior importância comparado com sistemas tradicionais. A análise e medidas de desempenho desses protocolos são uma rica fonte de conhecimento sobre o projeto interno de um sistema operacional moderno e de interesse pela comunidade acadêmica.

Esse trabalho teve os seguintes objetivos:

- Apresentar uma visão compreensiva do sistema Plan 9, especialmente sob seus aspectos distribuídos e dos mecanismos de comunicação suportados pelo kernel para comunicação tanto local quanto remota.

- Apesar de prover um ambiente inovador e bastante rico em termos de ferramentas de usuários (linguagens, ambientes para programação paralela, editores e depuradores) elas são apenas descritas no corpo do trabalho. O estudo detalhado do sistema concentrou-se nos aspectos de comunicação entre processos locais e remotos.

- A instalação e testes dos 3 componentes básicos do Plan 9 (terminais de usuário, servidor de CPU e servidor de arquivos) em ambiente PC também foi um dos objetivos do trabalho e envolveu a geração completa do sistema.

- Finalmente, a partir dos fontes em C e como objetivo maior, procurou-se medir e comparar o desempenho dos mecanismos de comunicação do kernel através de micro-benchmarks e de macro-benchmarks, tanto local quanto remotamente, nos sistemas operacionais Plan 9 e Linux , na mesma plataforma de hardware. Embora descendentes do UNIX esses sistemas são radicalmente diferentes em termos de projeto. Os benchmarks também foram executados numa máquina SPARC com o sistema UNIX/SunOS com o objetivo de comparar o desempenho do Plan 9 com UNIX/NFS.

  Para atingir esse objetivo um estudo da bibliografia recente de benchmarks foi feito, a fim de escolher benchmarks apropriados e que estivessem disponíveis publicamente.

1.2 Organização da dissertação

No final desse capítulo é feita uma breve descrição de dois sistemas operacionais distribuídos relacionados, o Amoeba e o Mach.

Os capítulos restantes desta dissertação estão organizados da seguinte forma:

O capítulo 2 apresenta uma visão geral do sistema Plan 9, desde sua decomposição funcional, idéias básicas, ferramentas disponíveis, protocolos de comunicação, portabilidade, e uma pequena descrição do modelo de programação paralela.

O capítulo 3 descreve em detalhes a arquitetura de recursos distribuídos do sistema Plan 9, apresentando o protocolo de comunicação do sistema de arquivos, 9P, e como os recursos estão distribuídos e podem ser acessados tanto local quanto remotamente. Esse capítulo apresenta também o protocolo IL, que é um novo protocolo de transporte usado no Plan 9.

O capítulo 4 mostra os procedimentos executados na instalação do sistema Plan 9 na arquitetura PC, mostrando também os principais problemas encontrados, e como estes foram solucionados.

O capítulo 5 é o corpo da tese: apresenta uma visão geral dos benchmarks usados nos testes de desempenho. Apresenta os resultados desses testes nos sistemas Plan 9, Linux, e SunOS, e conclusões baseadas nesses resultados.

O capítulo 6 apresenta as conclusões e os principais resultados obtidos nesta dissertação.

Por fim, o apêndice provê alguns conceitos básicos e terminologia sobre a área de sistemas operacionais e sistemas distribuídos.

1.3 Sistemas Operacionais relacionados

1.3.1 Amoeba

Amoeba[Tanenbaum90] é um sistema operacional distribuído desenvolvido na Universidade de Vrije em Amsterdam a partir de 1981 por Andrew Tanenbaum e seus estudantes de doutorado.

O modelo Amoeba

O modelo usado por Amoeba está ilustrado na Figura 1.1. Neste modelo toda a força computacional está localizada em um ou mais "pool de processadores". O segundo elemento dessa arquitetura é a estação de trabalho. É através das estações que os usuários acessam o sistema. Outros componentes importantes são os servidores especializados, tais como servidores de arquivos e os "gateways". A filosofia de distribuir serviços e funções do sistema operacional no Amoeba é muito semelhante à do Plan 9.

Objetivos e características

Amoeba tem três objetivos centrais.

- Transparência de rede
- Gerenciamento de recursos baseados em objetos e "capabilities"

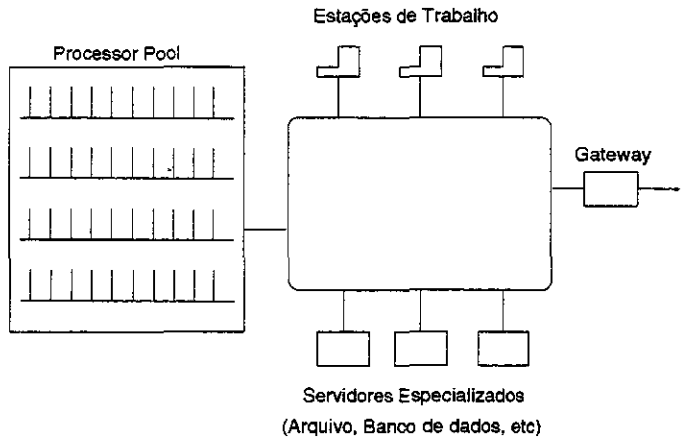


Figura 1.1: Arquitetura do sistema Amoeba

- Servidores no nível de usuário

O microkernel

O microkernel executa em todas as máquinas no sistema. Ele tem quatro funções primárias.

- Gerenciar processos e threads dentro desses processos (Figura 1.2a).
- Prover suporte de baixo nível de gerenciamento de memória
- Suporte transparente de comunicação entre threads arbitrárias (Figura 1.2b).
- Tratamento de E/S.

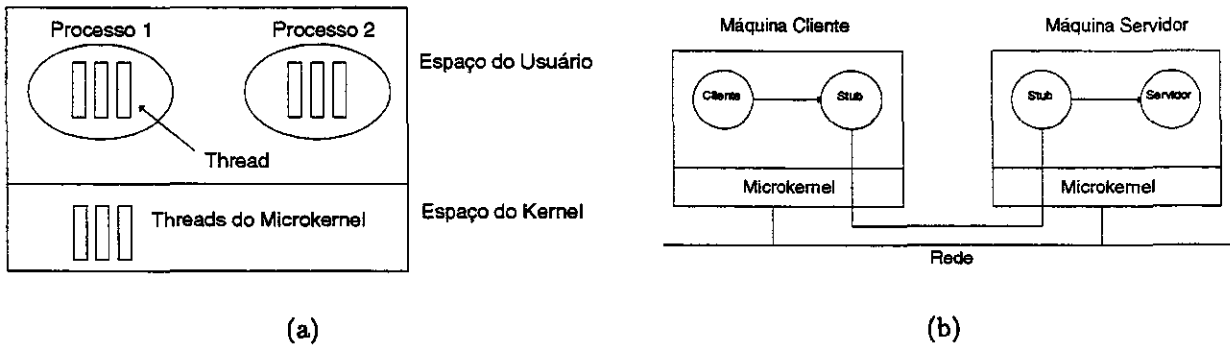


Figura 1.2: (a)Threads no Amoeba e (b)Chamada de Procedimento Remoto

Capabilities

No Amoeba todos os identificadores de recursos são capabilities, implementadas na forma mostrada na Figura 1.3. Uma capability tem 128 bits e contém um identificador que é mapeado em tempo de execução para uma porta de servidor ("server port"), e o número do objeto ("object number"), que é usado para identificar o objeto dentro do servidor. Os dois campos adicionais, o campo de permissões ("permissions field") e o campo de verificação ("check field"), são usados respectivamente para identificar os tipos de acessos que o possuidor da capability pode fazer, e para proteger a capability de falsificações.

Um dos problemas de desempenho do Amoeba é que todos os mecanismos de tratamento de capabilities são implementados por software.

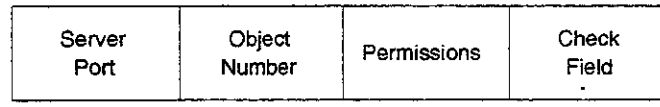


Figura 1.3: Uma capability do Amoeba

O Protocolo FLIP

FLIP é um protocolo sem conexão projetado para suportar transparência, chamadas eficientes de procedimentos remotos, comunicação em grupo, comunicação segura e facilidade de gerenciamento da rede.

1.3.2 Mach

O projeto Mach[Accetta86] foi desenvolvido na Universidade de Carnegie-Mellon. Em contraste a seus predecessores RIG e Accent e a Amoeba, o projeto Mach nunca se propôs a desenvolver um sistema operacional distribuído completo. Ele foi projetado para prover serviços avançados de kernel, que complementariam os do UNIX. No início, a intenção dos projetistas era que o UNIX fosse implementado como processos no nível de usuário.

A Figura 1.4 mostra a estrutura do Mach versão 3, a qual tem um *microkernel* muito pequeno.

Objetivos e características

Os principais objetivos e características do projeto são:

- *Operação em multiprocessador*: Mach foi projetado para executar em um multiprocessador de memória compartilhada, de maneira que tanto threads do kernel quanto threads do modo usuário pudessem ser executados por qualquer processador.

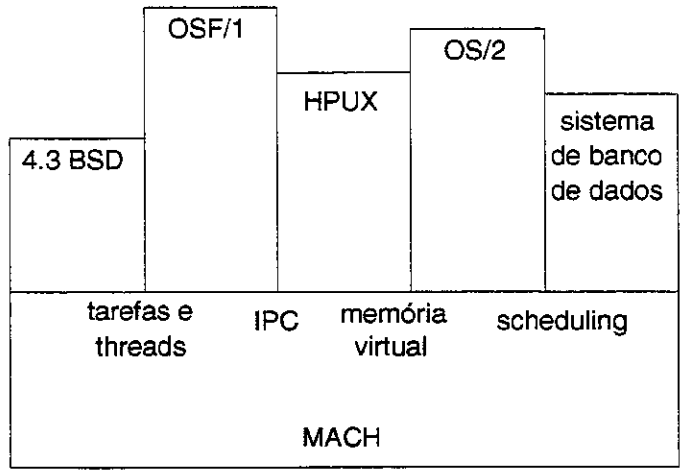


Figura 1.4: Estrutura do sistema Mach 3

- *Extensão transparente a operações de rede*: Para permitir que programas pudessem ser distribuídos transparentemente entre uniprocessadores e multiprocessadores através de uma rede, Mach adotou um modelo de comunicação independente de localização envolvendo portas como destinos.
- *Servidores no nível de usuário*: Mach suporta um modelo orientado a objetos no qual recursos são gerenciados ou pelo kernel ou por servidores no nível de usuário.
- *Emulação de sistema operacional*: Para suportar a emulação no nível binário do UNIX e outros sistemas operacionais, Mach permite o redirecionamento transparente de chamadas do sistema operacional para emulação de chamadas de biblioteca e conseqüentemente aos servidores do sistema operacional no nível de usuário (Figura 1.5).
- *Implementação de memória virtual flexível e portabilidade*: Mach foi projetado para ser portável para várias plataformas de hardware, com distintas arquiteturas de memória virtual, tendo sido extremamente bem sucedido nesse aspecto.

Principais abstrações

- *Tarefa*: é um ambiente de execução que provê a unidade básica de alocação de recursos. Uma tarefa consiste de um espaço virtual de endereços e acesso protegido ao recursos do sistema através de portas. Uma tarefa pode conter uma ou mais threads, e corresponder aproximadamente a um processo UNIX.

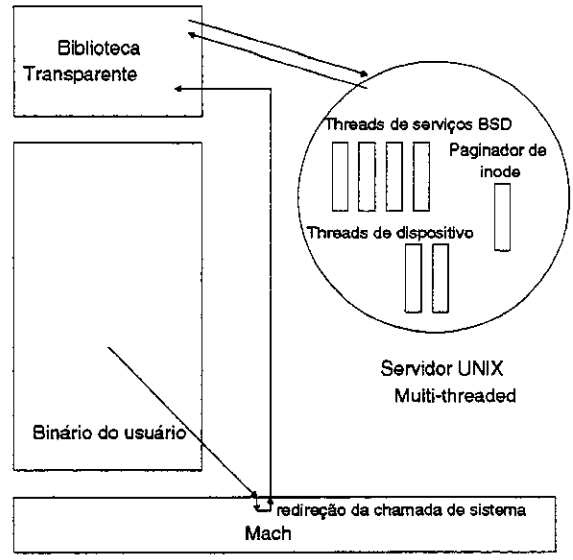


Figura 1.5: Mach

- *Thread*: é a unidade básica de execução, e deve executar no contexto de uma tarefa. Todas as threads dentro de uma tarefa compartilham os recursos da tarefa (portas, memória, etc). Um processo tradicional pode ser implementado como uma tarefa com uma única thread de controle.

- *Porta*: é o mecanismo básico de referência a um objeto em Mach, e é implementada como um canal de comunicação protegido pelo kernel. A comunicação é realizada através do envio de mensagens para as portas; mensagens são enfileiradas na porta destino se nenhuma thread está imediatamente pronta para recebê-las. Portas são protegidas por capabilities gerenciadas pelo kernel, ou *ports rights* ("direitos das portas"); uma tarefa deve ter um direito à porta para enviar uma mensagem para a porta. O programador invoca uma operação em um objeto enviando uma mensagem para uma porta associada com esse objeto. O objeto sendo representado por uma porta *recebe* as mensagens.

- *Conjunto de Portas*: é um grupo de portas compartilhando uma fila comum de mensagens. Uma thread pode receber mensagens para um conjunto de portas, e conseqüentemente servir múltiplas portas. Cada mensagem recebida identifica a porta individual (dentro do conjunto) a que se destina; o recebedor pode usar isto para identificar o objeto referido pela mensagem.

- *Mensagem*: é o método básico de comunicação entre threads no Mach. Ela é uma coleção de objetos de dados contidos; para cada objeto, ela pode conter os dados atuais ou um ponteiro para dados "out-of-line". Direitos de porta são passados em

mensagens; a passagem de direitos de porta em mensagens é a única maneira de movê-los entre tarefas. (A passagem de um direito de porta em memória compartilhada não funciona, pois o kernel do Mach não irá permitir que a nova tarefa use um direito obtido desta maneira.)

- *Objeto de memória*: é uma fonte de memória; tarefas podem acessá-lo através do mapeamento de porções (ou o objeto completo) em seu espaço de endereço. O objeto pode ser gerenciado por um *gerenciador de memória externo* em modo usuário. Um exemplo é um arquivo gerenciado por um servidor de arquivos; entretanto, um objeto de memória pode ser qualquer objeto para o qual o acesso a memória mapeada faz sentido. A implementação de um buffer de memória mapeada de um pipe UNIX é um exemplo.

  A Figura 1.6 ilustra essas abstrações.

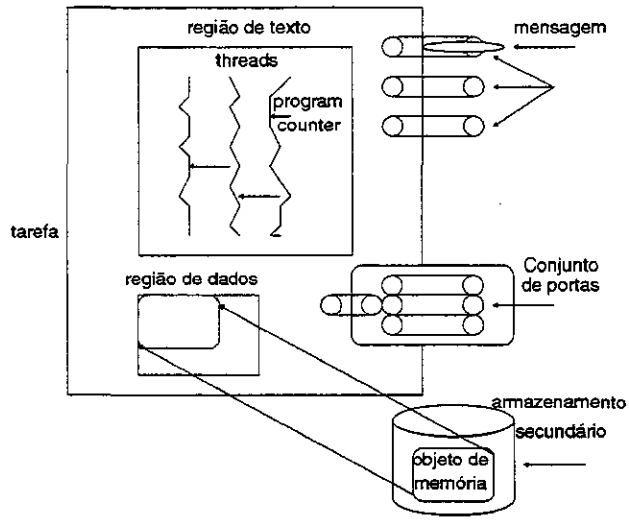


Figura 1.6: Abstrações básicas do sistema Mach

Suporte a redes no Mach

O Mach suporta redes como um processo de usuário (fora do kernel), facilitando a sua implementação e depuração. Posteriormente, verificou-se que esta é umas das maiores fontes de ineficiência do Mach. Plan 9 optou, apropriadamente, por incorporar o suporte aos protocolos de rede TCP/IP no próprio kernel.

Capítulo 2

Uma visão geral do sistema Plan 9

A idéia por trás do desenvolvimento do Plan 9 foi a de construir um sistema de tempo-compartilhado a partir de estações de trabalho, de uma maneira moderna, isto é, diferentes computadores teriam diferentes funções: máquinas baratas e pequenas serviriam como terminais provendo acesso a grandes recursos compartilhados e centrais como servidores computacionais e servidores de arquivos. Para as máquinas centrais, seriam usados multiprocessadores de memória compartilhada. A filosofia é muito parecida com aquela do Cambridge Distributed System[Needham82], que utiliza o conceito de "pool de processadores". O sistema deveria adaptar-se bem às mudanças no hardware e tirar proveito dos contínuos melhoramentos em máquinas pessoais, tais como vídeos gráficos bitmap, redes de alta e média velocidade e microcomputadores de alto desempenho. A equipe da Bell Laboratories queria que o sistema operacional fosse um sistema distribuído de propósito geral, multiusuário, portável e que pudesse ser implementado em uma variedade de arquiteturas de computadores e redes. O UNIX é um sistema de tempo-compartilhado que tem dificuldades de adaptar-se às idéias surgidas depois de sua concepção. No entanto, algumas idéias centrais vieram do UNIX ao invés de adicionar-se novos conceitos não experimentados.

O estilo de computação corrente oferece a cada usuário uma estação de trabalho ou PC dedicado. A abordagem do Plan 9 é diferente. As várias máquinas com vídeos, teclados e mouses provêm acesso aos recursos da rede, e devem ser funcionalmente equivalentes, iguais aos terminais conectados aos antigos sistemas de tempo-compartilhado. Quando alguém usa o sistema, entretanto, o terminal é temporariamente personalizado para aquele usuário. Ao invés de personalizar o hardware, Plan 9 oferece a capacidade para personalizar, via software, a visão do sistema pelo usuário. Essa personalização é realizada através da atribuição de nomes locais e pessoais para os recursos publicamente visíveis na rede. Plan 9 provê mecanismos para montar uma visão pessoal do espaço público com nomes locais para recursos acessíveis globalmente. Uma vez que os recursos

mais importantes da rede são arquivos, o modelo dessa visão é orientado a arquivos.

2.1 Decomposição da estrutura funcional

Plan 9 usa três tipos de componentes: terminais, que dão a cada usuário do sistema um computador dedicado, com um vídeo bitmap e mouse, no qual executa um sistema de janelas; servidores de arquivos (File Servers), que armazenam dados permanentes, servindo como repositórios de dados; servidores de CPU (CPU Servers) que concentram a força computacional em grandes processadores; e outros servidores que provêm autenticação de usuário e gateways de rede. Tipicamente usuários interagem com aplicações que executam tanto em terminais quanto em servidores de CPU, e as aplicações obtêm seus dados dos servidores de arquivos. Apesar do projeto ser altamente configurável, ele foge dos modelos específicos de estações de trabalho em rede e serviços de máquinas centrais. O compartilhamento de serviços de computação e o armazenamento de arquivos provê um sentido de comunidade para um grupo de programadores, amortiza custos, e centraliza e simplifica a administração. Esta arquitetura é reminescente da estrutura centralizada terminal-CPU de tempo compartilhado dos anos 70. Entretanto, Plan 9 tem algumas diferenças significativas.

Plan 9 acomoda uma variedade de redes diferentes. Os serviços comunicam-se por um único protocolo, construído sobre uma camada de transporte de dados confiável oferecida por uma rede apropriada. Mesmo para serviços não usualmente considerados como arquivos, o projeto unificado permite alguma simplificação digna de nota. Cada processo tem um espaço de nomes local que contém ligações a todos os serviços que o processo está usando e desse modo para os arquivos naqueles serviços. Uma das tarefas mais importantes de um terminal é suportar a visão personalizada do sistema representado pelos serviços ao usuário.

A Figura 2.1 mostra uma grande instalação Plan 9 onde servidores de CPU e servidores de arquivos compartilham redes locais de alta velocidade, enquanto terminais usam redes locais como Ethernet, Datakit ou redes de longa distância via linhas telefônicas. Máquinas "gateway" são servidores de CPU conectados a múltiplas redes, permitindo que usuários que trabalham em uma rede possam acessar outra.

Plan 9 explora três idéias básicas:

- Todos os objetos do sistema apresentam-se como arquivos nomeados que são manipulados por operações de leitura/escrita (read/write).
- Todos esses arquivos podem existir tanto local quanto remotamente e respondem a um protocolo padrão chamado 9P, que pode ser transmitido através de uma rede.

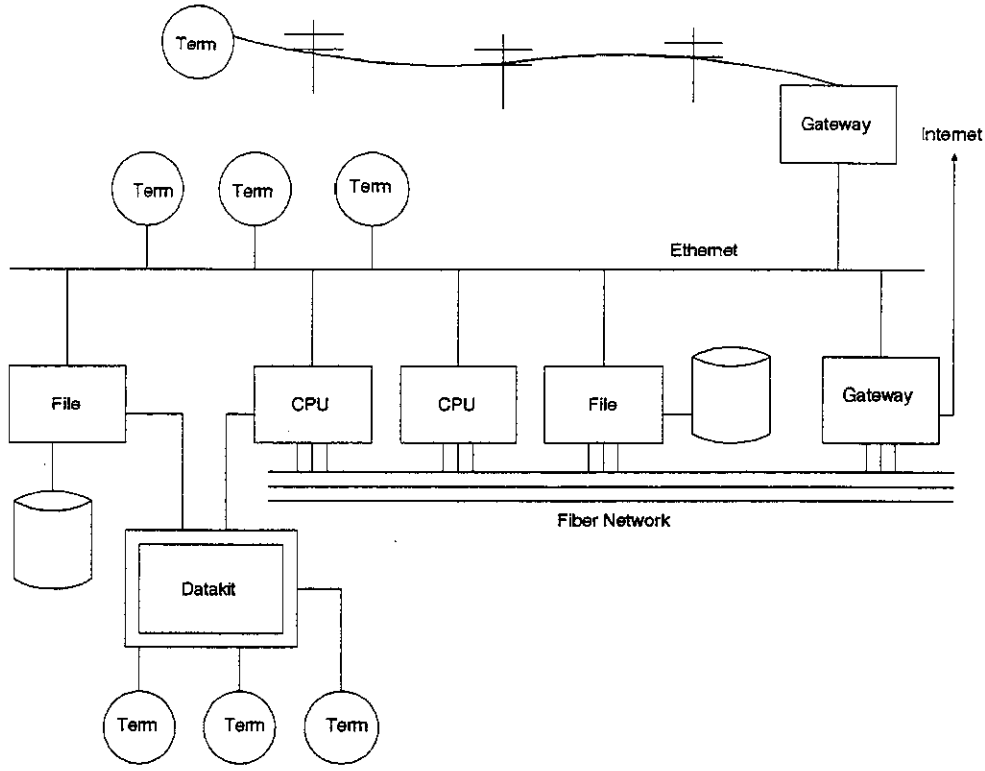


Figura 2.1: Estrutura de uma grande distribuição Plan 9.

Conseqüentemente todo o projeto é voltado a um sistema de arquivos remoto, não sendo mais necessário saber onde um arquivo particular reside.

- O espaço de nomes do sistema de arquivos, isto é, o conjunto de objetos visíveis a um programa, é dinamicamente e individualmente ajustável para cada um dos programas executando em uma máquina particular. Esta é uma idéia nova e tem efeitos interessantes. Essencialmente, significa que o conjunto de arquivos que o sistema operacional pode ver pode ser ajustado por processo. Conseqüentemente, programas de aplicação não precisam saber onde eles estão executando; onde e em qual tipo de máquina, é uma decisão econômica que não afeta a construção da própria aplicação. Tem como vantagens a melhoria da generalidade e a modularidade do projeto da aplicação por usar servidores que fazem qualquer tipo de informação apresentar-se aos usuários e às aplicações como coleções de arquivos ordinários.

O sistema operacional para os servidores de CPU e terminais é estruturado como um kernel tradicional: uma única imagem compilada contendo código para gerenciamento de recursos, controle de processos, processos do usuário, memória virtual e E/S. Devido ao fato de o servidor de arquivos ser uma máquina separada, o sistema de arquivos não está

compilado no kernel, embora esteja o gerenciamento do espaço de nomes, um atributo do processo.

2.2 A organização de redes no Plan 9

A arquitetura do sistema Plan 9 demanda uma hierarquia de velocidades de redes correspondendo às necessidades dos componentes. As conexões entre servidores de arquivos e servidores de CPU são de fibra ponto-a-ponto de alta largura de banda. Conexões dos servidores aos terminais locais usam redes de média velocidade tais como Ethernet e Datakit. Conexões de baixa velocidade através da Internet e do backbone da AT&T servem usuários em Oregon e Illinois. Serviços de dados ISDN de banda básica e linhas seriais de 9600 baud provêm conexões de baixa velocidade aos usuários em casa.

Uma vez que servidores de CPU e terminais usam o mesmo kernel, usuários podem escolher executar programas localmente em seus terminais ou remotamente nos servidores de CPU. A organização do Plan 9 oculta os detalhes do sistema de conectividade permitindo que usuários e administradores configurem seus ambientes para serem tão distribuídos ou centralizados quanto eles queiram. Comandos simples suportam a construção de espaço de nomes representados localmente, estendendo-se a muitas máquinas e redes. No trabalho, usuários tendem a usar seus terminais como estações de trabalho, executando programas interativos localmente e reservando os servidores de CPU para dados ou tarefas de computação intensiva tais como compilação e computação de jogos de xadrez. Em casa ou quando conectados através de uma rede de baixa velocidade, usuários tendem a fazer a maior parte do trabalho no servidor de CPU para minimizar o tráfego em conexões de baixa velocidade. O objetivo da organização da rede é prover o mesmo ambiente para o usuário onde quer que os recursos sejam usados.

Redes fazem um papel central em qualquer sistema distribuído. Isto é particularmente verdadeiro no Plan 9 onde a maior parte dos recursos é provida por servidores externos ao kernel. A importância do código de rede dentro do kernel é refletido por seu tamanho; de 25.000 linhas de código do kernel, 12.500 são para redes e protocolos relacionados. Redes estão continuamente sendo adicionadas ao Plan 9.

2.2.1 Modelo de Processos e Programação Paralela

Como a maioria dos sistemas operacionais, Plan 9 suporta o conceito de processos. Mas ao invés de seguir a tendência dos novos sistemas operacionais em implementar processos normais estilo UNIX e threads (ou processos leves) no kernel, Plan 9 provê uma única classe de processos, permitindo controle fino do compartilhamento dos recursos de um processo tais como memória e descritores de arquivos. Uma única classe de processos é

uma abordagem possível no Plan 9 pois o kernel tem uma interface eficiente de chamadas de sistema ("system calls") e criação de processos e escalonamento baratos. Os testes executados nos benchmarks confirmam parcialmente esta afirmação: criação de processos e chaveamento de processos são operações tão eficientes no Plan 9 quanto no Linux.

O suporte de Plan 9 para programação paralela tem dois aspectos. Primeiro, o kernel provê um modelo de processos simples e algumas chamadas de sistema cuidadosamente projetadas para sincronização e compartilhamento. Segundo, uma nova linguagem de programação paralela chamada Alef suporta programação concorrente. Embora seja possível escrever programas paralelos em C, Alef é a linguagem paralela preferencial.

Programas paralelos têm três requerimentos básicos: gerenciamento de recursos compartilhados entre processos, uma interface ao escalonador, e uma sincronização de processos refinada usando spin locks. No Plan 9, novos processos são criados usando a chamada de sistema *rfork*. *Rfork* toma como único argumento, um vetor de bits que especifica quais dos recursos do processo pai deverão ser compartilhados, copiados, ou criados novamente no filho. Os recursos controlados por *rfork* incluem o espaço de nomes, o ambiente, a tabelas de descritores de arquivos, segmentos de memória, e notas (análogo no Plan 9 aos sinais do UNIX). Um dos bits controlam se a chamada *rfork* irá criar um novo processo. Se o bit está desligado, a modificação resultante dos recursos ocorrem no processo fazendo a chamada. Por exemplo, um processo chama *rfork(RFNAMEG)* para desconectar seu espaço de nomes de seu pai. Alef usa um *fork* refinado no qual todos os recursos, incluindo memória, são compartilhados entre pai e filho, análogo a criar uma thread de kernel em muitos sistemas.

Uma indicação de que *rfork* é o modelo correto é a variedade de maneiras que ele é usado. Além do uso canônico na rotina de biblioteca *fork*, é difícil de encontrar duas chamadas para *rfork* com o mesmo conjunto de bits; programas o usam para criar muitas formas diferentes de compartilhamento e alocação de recursos. Um sistema com apenas dois tipos de processos - processos regulares e threads - não poderá tratar esta variedade.

Há duas maneiras de compartilhar memória. Primeiro, um flag para *rfork* faz com que todos os segmentos de memória do pai sejam compartilhados com o filho (exceto a pilha, a qual é sempre "forked copy-on-write"). Alternativamente, um novo segmento de memória pode ser conectado usando a chamada de sistema *segattach*. Tal segmento irá sempre ser compartilhado entre pai e filho.

A chamada de sistema *rendezvous* provê uma maneira de processos se sincronizarem. Alef a usa para implementar canais de comunicação, locks ("trava") de filas, locks de múltiplos leitores/escritores, e o mecanismo *sleep* e *wakeup*. *Rendezvous* toma dois argumentos, uma marca e um valor. Quando um processo chama *rendezvous* com uma marca ele dorme até que um outro processo apresente uma marca casando. Quando um par de marcas casam, os valores são trocados entre os dois processos e ambas as chamadas a

rendezvous retornam. Esta primitiva é suficiente para implementar o conjunto completo de rotinas de sincronização.

Finalmente, spin locks são providos por uma biblioteca dependente da arquitetura em nível de usuário. A maioria dos processadores provêm instruções *test and set* atômicas que podem ser usadas para implementar locks. Uma exceção notável é o MIPS R3000, de maneira que os multiprocessadores série SGI Power têm locks de hardware especiais no barramento. Processos do usuário obtêm acesso ao lock do hardware mapeando páginas de locks hardware em seu espaço de endereço usando a chamada de sistema *segattach*.

Um processo Plan 9 em uma chamada de sistema será bloqueado independentemente da sua importância. Isto significa que quando um programa deseja ler de um dispositivo lento sem bloquear todo o cálculo, ele deve criar um processo para fazer a leitura para ele. A solução é iniciar um processo satélite que faz a E/S e entrega a resposta para o programa principal através de memória compartilhada ou talvez um pipe. Isto parece oneroso mas funciona de maneira fácil e eficiente na prática; de fato, a maioria das aplicações interativas Plan 9 são escritas em C, como o editor de texto Sam[Pike87] e executam como programas multiprocessos.

O suporte do kernel para programação paralela no Plan 9 são algumas centenas de linhas de código portável; algumas primitivas simples permitem que os problemas sejam tratados claramente em nível do usuário. Embora as primitivas funcionem bem em C, elas são particularmente expressivas dentro de Alef. A criação e gerenciamento de processos de E/S escravos podem ser escritos em algumas linhas de Alef, provendo a fundação para um significado consistente de multiplexar fluxo de dados entre processos arbitrários. Mais ainda, implementando-o em uma linguagem ao invés de no kernel assegura semântica consistente entre todos os dispositivos e provê uma primitiva de multiplexação mais geral. Comparando isto com a chamada de sistema *select* do UNIX: *select* aplica-se somente a um conjunto restrito de dispositivos, legisla um estilo de multiprogramação no kernel, não se estende através de redes, é difícil de implementar, e é difícil de usar.

Uma outra razão por que programação paralela é importante no Plan 9 é que servidores de arquivos "multithreaded" no nível de usuário são a maneira preferida para implementar serviços. Exemplos de tais servidores incluem o ambiente de programação Acme[Pike94], a ferramenta para exportar espaço de nomes *exportfs*[Presotto93], o daemon HTTP, e os servidores de nome de rede *cs* e *dns*[Winterbottom93]. Aplicações complexas tais como Acme provam que o suporte cuidadoso de sistemas operacionais podem reduzir a dificuldade de escrever aplicações multi-threaded sem mover primitivas de threads e sincronização para dentro do kernel.

2.3 Portabilidade

Plan 9 é portável para uma variedade de arquiteturas de processadores. Dentro de uma única sessão de computação, é comum usar várias arquiteturas: por exemplo, o sistema de janelas executando em um processador Intel conectado a um servidor de CPU baseado em MIPS com arquivos residentes em um sistema SPARC. Para esta heterogeneidade ser transparente, deve existir convenções sobre a troca de dados entre programas; para a manutenção de software ser direta, deve existir convenções sobre compilações sob diferentes arquiteturas ("cross-architecture").

Para evitar problemas de ordem de bytes, dados são comunicados entre programas como texto. Algumas vezes, entretanto, a quantidade de dados é grande o suficiente que um formato binário se faz necessário; tais dados são comunicados como um stream de bytes com uma codificação pré-definida para valores multi-byte. Em raros casos onde o formato é complexo o suficiente para ser definido por uma estrutura de dados, a estrutura nunca é transmitida como uma unidade; ao invés disso, ela é decomposta em campos individuais, codificados como stream de bytes ordenados e então reunidos de novo pelo recipiente.

Programas, incluindo o kernel, freqüentemente apresentam seus dados através de uma interface de sistema de arquivos, um mecanismo de acesso que é inerentemente portável. Por exemplo, o relógio do sistema é representado por um número decimal no arquivo /dev/time; a função de biblioteca *time* (não existe chamada de sistema *time*) lê o arquivo e o converte para binário.

Cada arquitetura suportada tem seus próprios compiladores e carregadores ("loaders"). Os compiladores C e Alef produzem arquivos intermediários que são portavelmente codificados; os conteúdos são únicos para a arquitetura alvo mas o formato do arquivo é independente do tipo do processador da compilação. Quando um compilador para uma dada arquitetura é compilado em um outro tipo de processador e então usado para compilar um programa lá, o código intermediário na nova arquitetura é idêntico ao código intermediário no processador nativo. Do ponto de vista do compilador, toda compilação é produzida a partir de um compilador que executa em uma arquitetura, mas produz código para outra ("cross-compilation").

Embora cada carregador de uma arquitetura aceite somente arquivos intermediários produzidos por compiladores para essa arquitetura, tais arquivos podem ter sido gerados por um compilador executando em qualquer tipo de processador. Por exemplo, é possível executar o compilador MIPS em um 486, então usar o carregador MIPS em uma estação SPARC para produzir um executável MIPS.

Biblioteca ANSI/POSIX APE

Plan 9 vem com uma biblioteca que torna mais fácil importar aplicações que obedecem ao padrão POSIX. Existe também uma biblioteca que emula a interface de sockets de Berkeley. Para isso é provido um ambiente ANSI/POSIX conhecido como APE que combina um conjunto de cabeçalhos e bibliotecas de código objeto especificados pelo padrão ANSI C (ANSI X3.159-1989) com a interface padrão de sistemas operacionais POSIX (IEEE 1003.1-1990, ISO 9945-1), a parte de POSIX definindo as funções básicas de sistemas operacionais.

O comando *pcc* atua como um "front end" aos compiladores e carregadores do Plan 9. Ele executa um preprocessador ANSI C sobre arquivos fontes, usando os cabeçalhos APE para satisfazer diretivas #include <file>; ele então executa um compilador Plan 9; finalmente, ele pode carregar com bibliotecas APE para produzir um programa executável.

Executando-se o comando *ape/psh* inicia-se uma shell POSIX, com um ambiente que inclui os comandos POSIX *ar89*, *c89*, *cc*, *basename*, *dirname*, *expr*, *false*, *grep*, *kill*, *make*, *rmdir*, *sed*, *sh*, *stty*, *true*, *uname* e *yacc*.

2.4 Protocolos de Comunicação

O espaço de nomes local do cliente provê uma maneira de personalizar a visão da rede para o usuário. Todos os serviços disponíveis na rede exportam hierarquias de arquivos. Aqueles importantes para o usuário são reunidos em um espaço de nomes personalizado; aqueles sem interesse imediato são ignorados. Este é um estilo diferente da idéia de um "espaço de nomes uniforme global" como no sistema Sprite. No Plan 9 há nomes conhecidos para serviços comuns e nomes uniformes para arquivos exportados por esses serviços, mas a visão é completamente local.

Todos os recursos no Plan 9 assemelham-se a sistemas de arquivos. Isto não significa que eles são repositórios de arquivos permanentes em disco, mas que a interface para eles é orientada a arquivos (recursos tais como: dispositivos de E/S, serviços de backup, sistemas de janelas, interface de rede, etc) com as operações tradicionais, isto é, busca em uma árvore de nomes hierárquica, acesso a seus conteúdos através de chamadas read, write, etc. Há dúzias de tipos de sistemas de arquivos no Plan 9, mas poucos representam arquivos tradicionais. Neste nível de abstração, arquivos no Plan 9 são similares a objetos, exceto que arquivos são sempre providos de nomes, métodos de acesso, e métodos de proteção. Num sistema orientado a objetos, tais recursos deveriam ser criados para cada objeto.

A interface para o sistema de arquivos é definida por um protocolo, chamado 9P, que é análogo, embora não muito similar ao protocolo NFS[Sandberg85]. O protocolo 9P é estruturado como um conjunto de transações que enviam uma requisição de um

cliente para um servidor (local ou remoto) e retorna o resultado. 9P controla sistemas de arquivos, não apenas arquivos: dada uma conexão ao diretório raiz de um servidor de arquivos, as mensagens 9P navegam na hierarquia de arquivos, abrem arquivos para E/S, e lêem ou escrevem bytes arbitrários nos arquivos. Vê-se, portanto, que o acesso a arquivos é em nível de bytes e não de blocos, o que distingue 9P de protocolos como NFS e RFS.

2.4.1 O protocolo IL

O protocolo 9P deve executar sobre um protocolo confiável de transporte com garantia de entrega em seqüência e com mensagens delimitadas. 9P não possui mecanismos para recuperar-se de erros de transmissão e o sistema supõe que cada leitura de um canal de comunicação retornará uma mensagem 9P única. Pipes e alguns protocolos de rede já possuem essas propriedades, mas os protocolos padrão IP não. TCP tem um alto overhead e não preserva delimitadores. UDP embora barato e preservando delimitadores de mensagens, não provê entrega confiável e em seqüência.

Como nenhum protocolo reunia as características acima, um novo protocolo foi projetado satisfazendo:

- serviço confiável de datagrama
- entrega em seqüência
- internetworking usando IP
- baixa complexidade, alto desempenho
- timeouts ("temporizadores") adaptativos

IL(Internet Link) é um protocolo "lightweight" encapsulado por IP. Ele é baseado em conexão e provê transmissão confiável entre máquinas de mensagens seqüenciadas. Não há necessidade de controle de fluxo no IL uma vez que o protocolo é projetado para transportar mensagens RPC entre cliente e servidor, ou seja, um processo pode ter somente uma única requisição 9P pendente. Uma pequena janela para mensagens pendentes previne também muitas mensagens chegando de serem buferizadas; mensagens fora da janela são descartadas e devem ser retransmitidas.

2.5 Autenticação

Autenticação estabelece a identidade de um usuário acessando um recurso. O usuário requisitando o recurso é chamado cliente e o usuário concedendo o acesso ao recurso é

chamado servidor. Isto é usualmente feito sob os auspícios de uma mensagem 9P. Um usuário pode ser um cliente em uma troca ("exchange") de autenticação e um servidor em outra. Servidores sempre atuam em nome de algum usuário, tanto um cliente normal quanto alguma entidade administrativa, de modo que autenticação é definida para ser entre usuários, não entre máquinas.

Cada usuário Plan 9 tem uma chave de autenticação DES[NBS77] associada; a identidade do usuário é verificada pela capacidade de criptografar e descriptografar mensagens especiais chamadas challenges ("desafios"). Uma vez que o conhecimento de uma chave do usuário dá acesso a esses recursos do usuário, os protocolos de autenticação Plan 9 nunca transmitem uma mensagem contendo uma chave em sua forma legível (chamada "cleartext").

Autenticação é bilateral: no fim da troca de autenticação, cada lado está convencido da identidade do outro. Cada máquina inicia a troca com uma chave DES na memória. No caso de servidores de CPU e de arquivos, a chave, o nome do usuário, e o nome do domínio para o servidor são lidos de armazenamento permanente, usualmente RAM não-volátil (NVRAM). No caso de terminais, a chave é derivada de uma senha digitada pelo usuário no momento do boot. Uma máquina especial, conhecida como *servidor de autenticação*, mantém um banco de dados de chaves para todos os usuários em seu domínio administrativo e participa nos protocolos de autenticação.

O protocolo de autenticação atua da seguinte forma: após trocar challenges, uma parte contacta o servidor de autenticação para criar tickets de permissão de acesso criptografados com cada chave secreta da parte e contendo uma nova chave de comunicação. Cada parte descriptografa seu próprio ticket e usa a chave de comunicação para criptografar a outra challenge da parte.

Esta estrutura é similar ao sistema de autenticação Kerberos[Miller87], mas evita sua dependência de relógios sincronizados. Também ao contrário de Kerberos, a autenticação do Plan 9 suporta uma relação speaks for ("falar por")[Lampson91] que possibilita um usuário a ter a autoridade de outro: isto é como um servidor de CPU executa processos em nome de seus clientes.

A estrutura de autenticação do Plan 9 constrói serviços seguros em vez de depender de firewalls. Visto que firewalls requerem código especial para cada serviço ultrapassando a barreira ("wall"), a abordagem Plan 9 permite que a autenticação seja feita em uma único lugar - 9P - para todos os serviços. Por exemplo, o comando *cpu* funciona seguramente através da Internet.

2.5.1 Autenticando conexões externas

O protocolo de autenticação regular Plan 9 não é adequado para serviços baseados em texto como Telnet ou FTP. Em tais casos, usuários Plan 9 se autenticam com calculadores DES de mão chamados *autenticadores*. O autenticador armazena uma chave para o usuário, distinta da chave normal de autenticação do usuário. O usuário "loga no" autenticador usando um PIN (Personal Identification Number - Número de Identificação Pessoal) de 4 dígitos. Um PIN correto possibilita o autenticador para uma troca challenge/resposta com o servidor. Uma vez que uma troca challenge/reposta é válida somente uma vez e chaves nunca são enviadas através da rede, este procedimento não é suscetível a ataques de replay, mas é compatível com protocolos como Telnet e FTP.

2.5.2 Usuários especiais

Plan 9 não tem super-usuário. Cada servidor é responsável por manter sua própria segurança, usualmente permitindo acesso somente da console, a qual é protegida por uma senha. Por exemplo, servidores de arquivos têm um único usuário administrativo chamado *adm*, com privilégios especiais que aplicam-se somente a comandos digitados pela console física do servidor. Esses privilégios dizem respeito a manutenção diária do servidor, tais como adicionar novos usuários e configurar discos e redes. Os privilégios não incluem a capacidade de modificar, examinar, ou mudar as permissões de qualquer arquivo. Se um arquivo é protegido de leitura por um usuário, somente esse usuário pode dar acesso a outros.

Servidores de CPU têm um nome de usuário equivalente que permite acesso administrativo a recursos nesse servidor tais como arquivos de controle de processos do usuário. Tal permissão é necessária, por exemplo, para matar processos ociosos, mas não se estende além desse servidor. Por outro lado, por meio de uma chave armazenada em RAM não-volátil protegida, a identidade do usuário administrativo é provida ao servidor de autenticação. Isto permite que o servidor de CPU autentique usuários remotos, tanto para acesso ao próprio servidor quanto quando o servidor de CPU está atuando como um proxy em seu próprio nome.

Finalmente, um usuário especial chamado *none* não tem senha e é sempre permitido conectar; qualquer um pode afirmar ser *none*. *None* tem permissões restritas; por exemplo, ele não tem permissão para examinar dump de arquivos e pode ler somente arquivos públicos, ou seja, acessíveis para todo mundo.

A idéia por trás de *none* é análoga ao usuário anonymous ("anônimo") em serviços de FTP. No Plan 9, visitantes de servidores FTP são posteriormente confinados dentro de um espaço de nomes especial restrito. Ele desconecta usuários visitantes dos programas do sistema, tais como o conteúdo de */bin*, mas torna possível disponibilizar arquivos locais

para visitantes ligando-os ("binding") explicitamente dentro do espaço. Um nome de usuário restrito é mais seguro do que técnicas usuais de exportar uma árvore de diretórios ad hoc; o resultado é um tipo de proteção em volta de usuários não autorizados.

2.5.3 Autenticação proxy para serviço de cpu

Quando uma chamada é feita ao servidor de CPU por um usuário, digamos Peter, a intenção é que Peter deseja executar processos com sua própria autoridade. Para implementar esta propriedade, o servidor de CPU faz o seguinte quando a chamada é recebida. Primeiro, o listener ("escutador") cria um processo para tratar a chamada. Este processo muda para o usuário *none* para evitar conceder quaisquer permissões se ele está aceitando. Ele então executa o protocolo de autenticação para verificar que o usuário chamando é Peter, e para provar a Peter que a máquina é ela própria confiável. Finalmente ele se religa a todos os servidores de arquivos relevantes usando o protocolo de autenticação para identificar ele próprio como Peter. Neste caso, o servidor de CPU é um cliente do servidor de arquivos e executa a porção do cliente na mudança de autenticação em nome de Peter. O servidor de autenticação irá dar os tickets dos processos para realizar isto somente se o nome administrativo do usuário do servidor de CPU tem permissão para falar por Peter.

A relação *speaks for* é mantida na tabela no servidor de autenticação. Para simplificar o gerenciamento de computação dos usuários em diferentes domínios de autenticação, ele também contém mapeamentos entre nomes de usuários em domínios diferentes, por exemplo dizendo que o usuário *rtm* em um domínio é a mesma pessoa como o usuário *rtmorris* em outro.

2.5.4 Permissões de arquivos

Uma das vantagens de construir serviços como sistemas de arquivos é que as soluções para problemas de propriedade e permissão se encaixam naturalmente. Como no UNIX, cada arquivo ou diretório tem permissões de leitura, escrita e execução/busca separadas para o proprietário do arquivo, o grupo do arquivo, e quem quer que seja mais. A idéia de grupo não é usual: qualquer nome de usuário é potencialmente o nome de um grupo. Um grupo é apenas um usuário com uma lista de outros usuários no grupo. Convenções fazem a distinção: a maioria das pessoas têm nomes de usuários sem membros de grupo, enquanto grupos têm longas listas de nomes ligadas. Por exemplo, o grupo *sys* tradicionalmente tem todos os programadores do sistema, e os arquivos do sistema são acessíveis pelo grupo *sys*. Considere as seguintes duas linhas de um banco de dados do usuário armazenado em um servidor:

pjw:pjw
sys: :pjw,ken,philw,presotto

A primeira linha estabelece o usuário *pjw* como um usuário regular. A segunda estabelece o usuário *sys* como um grupo e lista quatro usuários os quais são membros desse grupo. A coluna vazia existente na segunda linha (entre *sys: :pjw*) é um espaço para um usuário ser nomeado como o líder do grupo. Se um grupo tem um líder, esse usuário tem permissões especiais para o grupo, tais como liberdade para mudar as permissões dos arquivos do grupo nesse grupo. Se nenhum líder é especificado, cada membro do grupo é considerado igual, como se cada um fosse o líder. No exemplo acima, somente *pjw* pode adicionar membros para este grupo, mas todos os membros de *sys* são parceiros iguais nesse grupo.

Arquivos regulares são de propriedade do usuário que os criou. O nome do grupo é herdado do diretório contendo o novo arquivo. Arquivos de dispositivos são tratados especialmente: o kernel pode agrupar a propriedade e permissões de um arquivo apropriado para o usuário que está acessando o arquivo.

Um bom exemplo da generalidade que isto oferece são arquivos de processos, os quais são possuídos e protegidos de leitura pelo proprietário do processo. Se o proprietário quer permitir que alguém mais acesse a memória de um processo, por exemplo para permitir o autor de um programa depurar uma imagem com erros, o comando padrão *chmod* aplicado a esses arquivos de processos faz a tarefa.

2.6 Ferramentas

Plan 9 vem com seus próprios compiladores para C e outras linguagens, junto com todas as ferramentas de programas e comandos originalmente disponíveis no ambiente UNIX. Ele também provê ferramentas novas de software.

2.6.1 Alef

Alef[Wint95] é uma linguagem de programação que provê mecanismos para implementar programas concorrentes. Um programa Alef parece-se com um programa em C. Apesar da similaridade da estrutura sintática e da sintaxe de expressão, a maior parte dos programas Alef são estruturalmente diferentes e funcionam diferentemente de programas em C.

Alef suporta dois modelos de sincronização de processos: variáveis compartilhadas e passagem de mensagens. Os estilos podem ser combinados livremente e a escolha freqüentemente depende de detalhes de projeto do programa. Programas que usam variáveis compartilhadas usam locks para sincronizar acessos aos dados compartilhados. O mode-

lo de passagem de mensagens usa as primitivas Alef que controlam, enviam, recebem e multiplexam canais.

2.6.2 Acme

Acme é um sistema híbrido de janelas, shell e editor que dá às aplicações orientadas a texto um estilo de orientação claro, expressivo e consistente. É uma ferramenta voltada para programadores. Sistemas tradicionais de janelas suportam interatividade com programas de clientes e oferecem bibliotecas de operações predefinidas tais como menus *pop-up* e botões para promover uma interface consistente com o usuário. Acme, ao contrário, oferece a seus clientes uma interface de usuário prefixada e convenções simples para motivar seu uso uniforme. Clientes acessam os serviços de Acme através de uma interface de sistema de arquivos; Acme é em parte um servidor de arquivo que exporta arquivos como dispositivos que podem ser manipulados para acessar e controlar os conteúdos de suas janelas. Escrito em Alef, Acme é estruturado como um conjunto de processos cooperantes que cuidadosamente subdividem os vários aspectos de suas tarefas: gerenciamento de vídeo, entrada, servidor de arquivo etc.

2.6.3 Acid

Acid[Winterbottom94] é um depurador simbólico no nível de programa fonte. O depurador é implementado como uma linguagem de comandos distinta da linguagem do programa sendo depurado. Essa linguagem oferece uma interface flexível com o usuário permitindo personalização para uma aplicação ou arquitetura específica. Embora seja requerido algum empenho para aprender a linguagem de depuração, a riqueza e a flexibilidade do ambiente de programação motivam novas maneira de raciocínio sobre a maneira com que os programas executam e as condições sob as quais eles falham. Acid é capaz de depurar múltiplos processos permitindo que eles compartilhem um conjunto comum de símbolos, como por exemplo, processos em um programa Alef.

Como outras soluções baseadas em linguagens, Acid apresenta uma interface pobre mas oferece uma ferramenta de depuração poderosa, ao contrário dos depuradores atuais.

Acid permite que a execução de um programa seja controlada através de operações em seu estado enquanto ele está parado, e através do monitoramento e controle de sua execução quando ele está executando. Cada ação do programa que cause a mudança do estado de execução é refletido pela execução de uma função Acid, que pode ser definida pelo usuário.

2.6.4 O editor de textos Sam

Sam é um editor interativo de textos multiarquivos para ser usado em vídeos bitmap, que combina edição interativa do tipo cortar-e-colar ("cut-and-paste") com uma linguagem de comandos baseada na composição de expressões regulares que descrevem a estrutura do texto sendo modificado.

Sam é implementado como dois processos, um tratando o vídeo e o outro os algoritmos de edição. Conseqüentemente ele pode executar com o processo vídeo em um terminal bitmap e o editor em um computador central (host), ou com ambos os processos em um host bitmap, ou com o processo vídeo no terminal e o editor em um host remoto. O processo vídeo pode também executar sem um terminal bitmap.

2.6.5 O sistema de janelas $8\frac{1}{2}$

Uma das características do Plan 9 é que ele foi projetado para ser usado em uma máquina com vídeo executando um sistema de janelas. Ele não tem noção de "teletype" como o UNIX. O tratamento de teclado do sistema é rudimentar.

O sistema de janelas do Plan 9 chamado $8\frac{1}{2}$[Pike91] é um programa de tamanho modesto e de projeto moderno. Ele provê serviços de E/S textual e gráficos bitmap tanto para o cliente local quanto para o remoto, oferecendo um serviço multiplexado de arquivos para esses clientes. Logo, uma vez que o sistema está executando, textos podem ser editados com aplicações "cortar-e-colar" de um menu pop-up, copiados entre janelas etc.

Cada janela é criada em um espaço de nomes separado. Ajustes feitos ao espaço de nomes em uma janela não afetam outras janelas ou programas. Essa janela também tem um bitmap privado e acesso ao teclado, mouse e outros recursos gráficos através de arquivos como /dev/mouse, /dev/bitblt e /dev/cons. Esses arquivos são fornecidos pelo $8\frac{1}{2}$, que é implementado como um servidor de arquivos. Ao contrário do X Windows, onde uma nova aplicação tipicamente cria uma nova janela para executar, uma aplicação gráfica $8\frac{1}{2}$ geralmente executa na janela onde ela é iniciada.

Capítulo 3

Arquitetura do Sistema Plan 9

Todos os recursos do Plan 9 que um processo pode acessar residem em um espaço de nomes e são acessados uniformemente. Esses recursos assemelham-se a sistema de arquivos e, doravante, serão chamados sistemas de arquivos. Isso não significa que eles são repositórios de arquivos permanentes em disco, mas que a interface para eles é orientada a arquivos: encontrar recursos em uma árvore de nomes hierárquica, ligá-los pelo nome, e acessar seus conteúdos através de chamadas de leitura e escrita (*read* e *write*). Sistemas de arquivos podem também ser do tipo tradicional com armazenamento persistente em disco como implementado pelo servidores de arquivos compartilhados. Arquivos também representam dispositivos físicos tais como terminais ou abstrações complexas tais como processos. Os sistemas de arquivos podem ser implementados por drivers residentes no kernel, por processos a nível do usuário, ou por servidores remotos, como mostrado na Figura 3.1 abaixo:

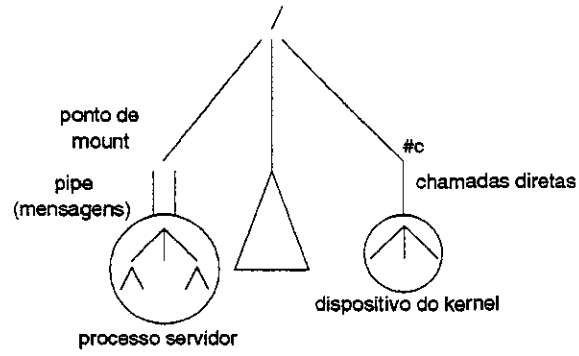


Figura 3.1: Espaço de nomes do Plan 9

Um sistema de arquivos representando um dispositivo físico normalmente contém dois arquivos. Por exemplo, uma linha RS232 é representada como um diretório contendo um arquivo *ctl* e um arquivo *data*. O arquivo *data* é um stream de bytes transmitidos/recebidos

na linha. O arquivo *ctl* é um canal de controle usado para alterar parâmetros do dispositivo tais como taxa de transmissão.

Muitas funções que requerem uma chamada de sistema em outros sistemas operacionais são representadas por operações de E/S sobre arquivos no Plan 9: ler o identificador de um processo, o identificador do usuário associado com um processo, etc.

3.1 O Protocolo 9P

O protocolo 9P contém 17 tipos de mensagens: três para inicializar e autenticar uma conexão e quatorze para manipular objetos. As mensagens 9P estão listadas na Tabela 3.1. Elas são geradas pelo kernel em resposta às requisições de E/S no nível do usuário ou do kernel. As mensagens *auth* e *attach* são usadas para obter um canal novo e autenticado no servidor. Os canais iniciam logicamente conectados à raiz da árvore do servidor. As mensagens *clone*, *walk*, e *clwalk* são usadas para travessia de "pathnames". *Clone* duplica um canal, e *walk* move um canal para um objeto com nome diferente. *Clwalk* combina essas operações. Finalmente, há as operações em objetos individuais: *open*, *read*, *write*, *clunk*(close), *create*, *stat* e *wstat*. Nenhuma das mensagens 9P consideram "cacheamento"; arquivos cacheados são providos, quando necessário, tanto dentro do servidor (cacheamento centralizado) ou implementados como um sistema de arquivos transparente entre o cliente e a conexão 9P ao servidor (cacheamento no cliente). Ou seja, o protocolo 9P não tem suporte explícito para fazer cacheamento de arquivos em um cliente. Isto ficou demonstrado nos benchmarks envolvendo transferências de arquivos, onde Plan 9 teve um desempenho nitidamente inferior a Linux. A memória do servidor de arquivos central, em geral grande, atua como um cache compartilhado por todos seus clientes, o que reduz a quantidade total de memória necessária nas máquinas da rede.

A conexão ao servidor é um caminho de comunicação bidirecional do cliente ao servidor. Pode haver um único cliente ou vários clientes compartilhando a mesma conexão. Uma árvore de arquivos do servidor é então ligada ao espaço de nomes do grupo do processo por chamadas *bind* e *mount*. Processos no grupo são então clientes dos servidores: chamadas de sistema operando sobre arquivos são traduzidas em requisições e respostas 9P transmitidas na conexão ao serviço apropriado.

Um cliente transmite *requisições* (mensagens T) para um servidor, o qual subseqüentemente retorna *respostas* (mensagens R) para o cliente. A ação combinada de transmitir uma requisição de um tipo particular, e receber sua resposta é chamada uma *transação* desse tipo. Mensagens são transportadas em bytes para permitir a independência de máquina.

Operação	Descrição
Nop	A operação nula
Session	Inicia uma sessão de comunicação
Error	Retorna uma mensagem de erro em resposta a outras mensagens
Flush	Aborta uma requisição de mensagem pendente
Auth	Autentica um cliente
Attach	Identifica o espaço de nomes do usuário e do servidor para ser acessado através do canal
Clone	Duplica uma referência a um objeto
Clunk	Fecha uma referência a um objeto
ClWalk	Combina as operações Clone e Walk
Open	Prepara a referência a um objeto para operações de E/S futuras
Create	Cria um objeto e o prepara para operações de E/S
Read	Lê dados de um objeto
Write	Escreve dados para um objeto
Remove	Remove um objeto de um servidor
Stat	Busca os atributos de um objeto
Wstat	Escreve os atributos de um objeto
Walk	Caminha através de uma hierarquia de diretórios

Tabela 3.1: Descrição do Protocolo 9P

3.2 Espaço de nomes

Os vários serviços usados por um processo são agrupados no espaço de nomes do processo, que é uma hierarquia única de nomes de arquivos, ou seja, o espaço de nomes é local para cada processo. Quando um processo cria um processo filho, este compartilha o espaço de nomes com o pai, significando que a visão local do mundo, para cada processo, pode ser compartilhada com outros processos. Processos do usuário constroem espaços de nomes usando três chamadas de sistema: *mount*, *bind*, e *unmount*. Antes de chamar *mount*, o cliente deve adquirir uma conexão ao servidor na forma de um descritor de arquivo que pode ser escrito e lido para transmitir mensagens 9P. Esse descritor de arquivo representa um pipe ou conexão de rede. A chamada *mount* conecta uma nova hierarquia ao espaço de nomes existente, ou seja, conecta uma árvore de arquivos do servidor ao espaço de nomes corrente (Figura 3.2).

Uma vez que a árvore de arquivos está conectada, seus membros podem ser acessados da mesma maneira como arquivos locais (Figura 3.3). Aplicações não podem detectar que esses arquivos são remotos. Isto é verdade mesmo para o arquivoC, um arquivo de um outro servidor que este cliente tem mapeado em seu espaço de nomes.

A chamada mount tem a seguinte forma:

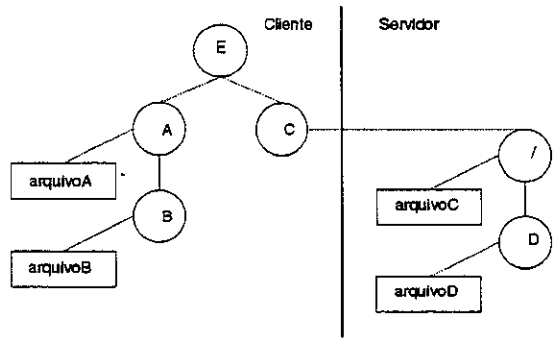


Figura 3.2: Visão cliente/servidor antes do mount

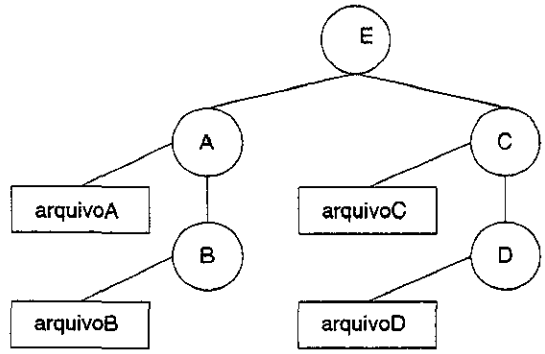


Figura 3.3: Visão do cliente após o mount

mount(int fd, char *old, int flags)

Mount autentica o usuário e conecta a árvore de arquivos do serviço para o diretório nomeado por *old*. A chamada de sistema *bind*, por outro lado, duplica algum pedaço de espaço de nomes existente em um outro ponto no espaço de nomes. Isto é, a chamada *bind* coloca um componente do espaço de nomes montado em uma localização desejada. A operação *bind* mapeia um arquivo de um espaço de nomes para outro. Por exemplo, uma aplicação pode esperar encontrar um certo arquivo, *arquivoB*, no diretório D, mas o arquivo na verdade reside no diretório B. Figura 3.3.

A operação *bind* pode ser usada para fazer com que o *arquivoB* apresente-se como residindo no diretório D. Figura 3.4.

Pode-se notar que *arquivoB* ainda está disponível no diretório B. Neste caso, os conteúdos originais do diretório D estão obscurecidos, pois operações de busca de arquivos procedem da esquerda para a direita. (Pode notar-se que os conteúdos do diretório D não são substituídos e eles estarão disponíveis novamente após um *unmount*. Opções *bind* diferentes podem ser usadas para tornar D um "diretório união" que tem tanto os conteúdos originais e aqueles do diretório B. "Diretórios união" são um conceito importante

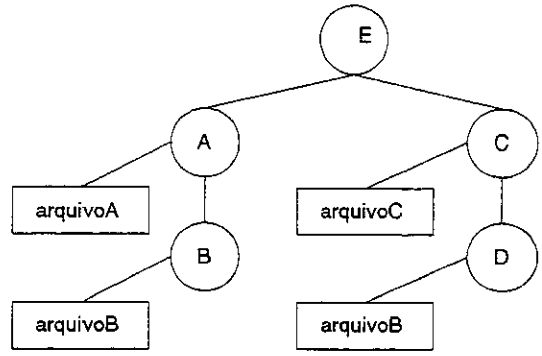


Figura 3.4: Visão dos arquivos após a operação bind

e inovador do Plan 9 a ser visto a seguir.

Bind tem a seguinte forma:

bind(char *new, char *old, int flags)

Bind toma a porção do espaço de nomes existente visível em *new*, tanto um arquivo quanto um diretório, e o torna também visível em *old*. Usando *bind* ou *mount*, múltiplos diretórios podem ser colocados no mesmo ponto no espaço de nomes. Na terminologia Plan 9, isto é um "diretório união" e comporta-se como a concatenação dos diretórios constituintes. O argumento *flag* em *bind* e *mount* especifica a posição do novo diretório na união, permitindo que novos elementos sejam adicionados no início ou fim da união ou que sejam substituídos completamente (Figura 3.5). "Diretórios união" são uma das características organizacionais do espaço de nomes do Plan 9 mais largamente usadas. Por exemplo, o diretório */bin* é construído como uma união de */cputype/bin* (binários dos programas), */rc/bin* (scripts do shell), e talvez mais diretórios providos pelo usuário. Esta construção torna a variável PATH do shell desnecessária.

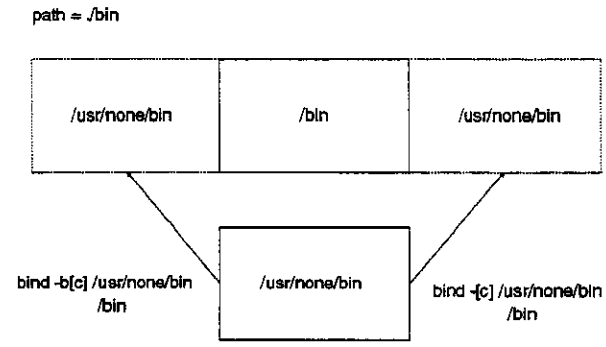


Figura 3.5: Alterando o espaço de nomes com bind

A instrução

bind -b A/B C/D

resulta em uma união dos conteúdos de dois diretórios, como mostrado na Figura 3.6.

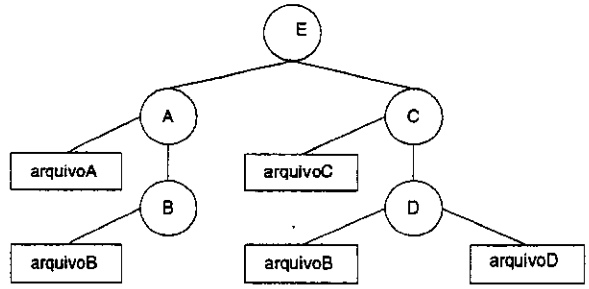


Figura 3.6: União dos conteúdos dos diretórios A/B e C/D

3.3 Servidores básicos do Plan 9

O servidor *exportfs* é um processo no nível do usuário que toma uma porção de seu próprio espaço de nomes e o torna disponível para outros processos através da tradução de requisições 9P em chamadas de sistema do kernel. A hierarquia que ele exporta pode conter arquivos de múltiplos servidores. *Exportfs* é usualmente executado como um servidor remoto inicializado por um programa local: *import* ou *cpu*.

Exportfs é invocado por uma chamada de rede. O listener (o equivalente Plan 9 de *inetd*) executa o perfil do usuário requisitando o serviço para construir um espaço de nomes antes de inicializar *exportfs*. Depois que um protocolo inicial estabelece a raiz da árvore de arquivos sendo exportada, o processo remoto monta a conexão, permitindo que *exportfs* atue como um servidor de arquivos de retransmissão. Operações na árvore de arquivos importada são executadas no servidor remoto e os resultados retornados. Como um resultado o espaço de nomes da máquina remota parece ser exportada em uma árvore de arquivos local.

Import faz uma chamada de rede para uma máquina remota, inicia *exportfs* nessa máquina, e liga sua conexão 9P ao espaço de nomes local, ou seja, ele conecta um pedaço do espaço de nomes de um sistema remoto ao espaço de nomes local e sai. Nenhum processo local é requerido para servir mounts; mensagens 9P são geradas pelo mount driver do kernel e enviadas diretamente através da rede. A simplicidade das interfaces encoraja usuários novatos a explorarem o potencial de um ambiente ricamente conectado, pois usando essas ferramentas é fácil a conexão entre redes. Por exemplo, *import helix*

/net torna as interfaces de rede da máquina Helix visíveis no diretório local da máquina que está fazendo a chamada *import*, no diretório */net*. Helix é um servidor central e tem muitas interfaces de rede, então isto permite que a máquina com uma rede acesse quaisquer redes de Helix. Após o *import*, a máquina local pode fazer chamadas em quaisquer das redes conectadas à Helix. Um outro exemplo é *import helix /proc* o qual torna os processo de Helix visíveis no diretório local */proc*, permitindo que depuradores locais examinem processos remotos.

Exportfs deve ser multithreaded uma vez que as chamadas de sistema *open*, *read* e *write* podem bloquear. Plan 9 não implementa a chamada de sistema *select* mas permite processos a compartilharem descritores de arquivo, memória e outros recursos. *Exportfs* e espaço de nomes configuráveis provêm um meio de compartilhamento de recursos entre máquinas. Ele é um bloco de construção para construir espaços de nomes complexos servidos a partir de muitas máquinas.

O comando *cpu* conecta o terminal local ao servidor de CPU remoto. Ele funciona em direção oposta à *import*: após chamar o servidor, ele inicia um *exportfs* local e o monta no espaço de nomes de um processo, tipicamente uma nova shell, no servidor. Ele então reorganiza o espaço de nomes para tornar arquivos de dispositivos locais (tais como aqueles servidos pelo sistema de janelas do terminal) visíveis no diretório */dev* do servidor. O efeito de executar um comando *cpu* é conseqüentemente iniciar uma shell em uma máquina rápida, mais fortemente acoplada ao servidor de arquivos, com um espaço de nomes análogo ao espaço de nomes local. Todos os arquivos de dispositivos locais são visíveis remotamente, logo aplicações remotas têm completo acesso aos serviços locais tais como vídeo gráfico bitmap, */dev/cons* etc. Os diretórios montados em */bin* são alterados para serem aqueles que contêm executáveis para o tipo do processador do servidor de CPU (o terminal pode ser um 68020 enquanto o servidor de CPU pode ser um MIPS). Isto não é o mesmo que *rlogin*, o qual nada faz para reproduzir o espaço de nomes local no sistema remoto, nem é o mesmo como compartilhar arquivos como, por exemplo, NFS, o qual pode obter equivalência de espaço de nomes mas não a combinação de acesso a dispositivos de hardware locais, arquivos remotos e recursos de CPU remotos. O comando *cpu* é um mecanismo transparente distintamente diferente de outros sistemas distribuídos. Por exemplo, é razoável iniciar um sistema de janelas em uma janela executando um comando *cpu*; todas as janelas criadas automaticamente iniciam processos no servidor de CPU. Em geral, um usuário digitando o comando *cpu* nota apenas que compilações ficam mais rápidas enquanto operações gráficas ficam mais lentas.

Import, *exportfs* e *cpu* ilustram muito bem o aspecto distribuído do Plan 9.

3.3.1 Servidor do kernel

Um dispositivo ("device") Plan 9 é um servidor no kernel que implementa uma árvore de arquivos para processos clientes. Um nome de arquivo começando com um caractere #, tal como #c, nomeia a raiz da árvore de arquivos implementada por um driver de dispositivo do kernel identificado pelo caractere após o #. Tais nomes são usualmente acoplados a localizações convencionais no espaço de nomes. Por exemplo, após *bind("#c", "/dev", MREPL)* um *ls* em */dev* irá listar os arquivos providos pelo dispositivo console.

Para maior eficiência, a conexão a sistemas de arquivos residentes no kernel é realizada através de chamadas de procedimentos locais ao invés de remotas, o que evita a sobrecarga do protocolo 9P. Uma vez que drivers do dispositivo são diretamente endereçáveis, não há necessidade de passar mensagens para se comunicar com eles; ao invés disso, cada transação 9P é implementada por uma chamada direta de procedimento. Para cada descritor, o kernel mantém uma representação local em uma estrutura de dados chamada canal, a qual é usada como um ponteiro para um arquivo. Um descritor de arquivo (fid) no nível do usuário é apenas um número que identifica um canal do kernel. Assim todas as operações executadas pelo kernel nos arquivos envolvem um canal conectado a esse *fid*. O exemplo mais simples é um arquivo de descritores dos processos do usuário, os quais são índices em um array de canais. Cada dispositivo contém um procedimento para cada uma das mensagens 9P, sendo que esse relacionamento um para um entre procedimentos e mensagens 9P são armazenados em uma tabela chamada *devtab*, ou seja, existem 17 ponteiros por driver, uma para cada tipo de mensagem 9P. Cada canal contém um deslocamento ("offset") em *devtab* indicando o driver a ser usado quando o arquivo que ele aponta é acessado. Uma chamada ao sistema tal como *read*, vinda do usuário, traduz-se em uma ou mais chamadas de procedimentos através dessa tabela. Cada chamada toma no mínimo um canal como argumento.

Quando a uma chamada de sistema é passado um nome de arquivo começando com # ela olha no próximo caractere, e se esse é um dispositivo válido ela executa uma conexão ao dispositivo correspondente para obter um canal representando a raiz dessa árvore de arquivos do dispositivo. Se há quaisquer caracteres após o caractere do dispositivo mas antes do próximo / ou fim da cadeia de caracteres, esses caracteres são passados como parâmetros para a conexão. Por exemplo, #Itcp identifica a implementação do protocolo TCP fornecido pelo dispositivo IP.

O acesso a sistemas de arquivos não residentes no kernel é realizado através de um driver de dispositivo especial, o "mount driver" (#M). Todos os canais apontando para este driver contêm um ponteiro para um canal de comunicação. O mount driver traduz chamadas de procedimentos em mensagens, isto é, ele converte chamadas de procedimentos locais 9P em mensagens RPC para serviços remotos através de um protocolo de transporte provido separadamente, tal como TCP ou IL, ou através de um pipe para um

processo do usuário. Chamadas *write* e *read* transmitem as mensagens através da camada de transporte. O mount driver é a única ponte entre a interface procedural vista pelo programa do usuário e serviços remotos no nível de usuário. Ele faz todo o "marshaling", gerenciamento de buffer e multiplexação associadas e é o único mecanismo integral de RPC no Plan 9. De fato, este driver especial torna-se um "proxy" local para os arquivos servidos por um servidor de arquivos remoto. Não há compilador stub RPC; ao invés disso o mount driver e todos os servidores compartilham apenas uma biblioteca que empacota e desempacota mensagens 9P (Figura 3.7).

A cadeia #M é um nome de arquivo ilegal, de maneira que este dispositivo pode somente ser acessado diretamente pelo kernel.

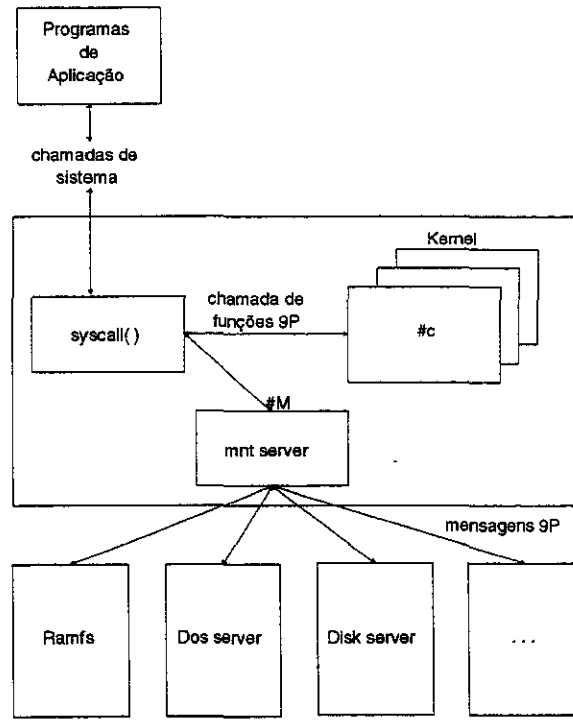


Figura 3.7: Mount device

A representação no kernel do espaço de nomes é chamada de "mount table", a qual armazena uma lista de ligações ("bindings") entre canais. Cada entrada na mount table contém um par de canais, um para comunicação saindo e outro para comunicação entrando.

Cada arquivo no Plan 9 é identificado univocamente por um conjunto de inteiros: o tipo do canal (usado como índice da tabela de chamadas de funções), o número do servidor ou dispositivo distinguindo o servidor de outros do mesmo tipo (decidido localmente pelo driver), e um *qid* formado de dois números de 32 bits chamados *path* e *versão*. O "path"

é um número unívoco de arquivo atribuído por um driver de dispositivo ou servidor de arquivo quando o arquivo é criado. O número da versão é atualizado quando o arquivo é modificado, e que é usado para manter a coerência de cache entre clientes e servidores.

O tipo e o número do dispositivo são análogos aos números de dispositivo, no UNIX, *major* e *minor*; o *qid* é análogo ao *i-number*. O dispositivo e o tipo conectam o canal ao driver do dispositivo e o *qid* identifica o arquivo dentro desse dispositivo.

3.4 Exemplo de serviço: "sistema de arquivo de processos"

Um exemplo de um serviço local é o "sistema de arquivos de processos", *proc*, que permite inspeção e depuração de processos executando através de uma interface orientada a arquivos. Ela é relacionada ao sistema de arquivos de processos de [Killian84] mas suas diferenças exemplificam a maneira como serviços Plan 9 são construídos.

O dispositivo *proc* serve uma estrutura de diretórios de dois níveis. O primeiro nível contém diretórios numerados correspondendo aos identificadores de processos (pids) dos processos vivos; cada um desses subdiretórios contém um conjunto de arquivos representando o processo correspondente.

mem — contém a imagem de memória corrente do processo. Um *read* ou *write* no offset *o*, que deve ser um endereço virtual válido, acessa bytes a partir do endereço *o* para o fim do segmento de memória contendo *o*. A memória virtual do kernel, incluindo a pilha do kernel para o processo e registradores salvos do usuário (cujos endereços são dependentes da máquina), podem ser acessados através de *mem*. Escritas são permitidas somente enquanto o processo está no estado *Stopped* e somente para os endereços ou registradores do usuário.

proc — contém uma estrutura de kernel por processo. Seu principal uso é para recuperar a pilha do kernel e o contador do programa (program counter) para depuração do kernel.

segment — contém um display textual dos segmentos de memória incorporados ao processo.

status — contém uma seqüência de caracteres com oito campos, cada qual seguido por um espaço. Os campos são: nome do processo, nome do usuário, estado do processo, quantidade de memória usada pelo processo, exceto sua pilha.

texto — como no UNIX, é um pseudônimo para o arquivo de código do processo; seu principal uso é recuperar a tabela de símbolos do processo.

wait — pode ser lido para recuperar registros *Waitmsg* dos filhos do processo. Se o processo não tem filhos existentes, vivos ou que saíram (exited), um *read* de *wait* irá bloquear.

ctl — Mensagens textuais escritas para esse arquivo controlam a execução do processo. Algumas requerem que o processo esteja em um estado particular e retornam um erro se ele não estiver.

- stop — Suspende a execução do processo, colocando-o no estado *Stopped*.
- start — Reinicia a execução de um processo *Stopped*.
- waitstop — Não afeta o processo diretamente mas, como todas as outras mensagens terminando com *stop*, bloqueia o processo que está escrevendo no arquivo *ctl* até que o processo alvo esteja no estado *Stopped* or exits ("saída"). Também como outras mensagens de controle *stop*, se o processo alvo irá receber uma nota enquanto a mensagem está pendente, ela é ao invés disso parada e o processo de depuração é reiniciado.
- startstop — Permite que um processo *Stopped* continue, e então faz uma ação *waitstop*.
- hang — Ativa um bit no processo de maneira que, quando ele termina uma chamada de sistema *exec*, ele irá entrar no estado *Stopped* antes de retornar ao modo usuário. Este bit é herdado através de um *fork*.
- nohang — Limpa o bit hang.
- kill — Mata o processo a próxima vez que ele ultrapassa o limite usuário/kernel.

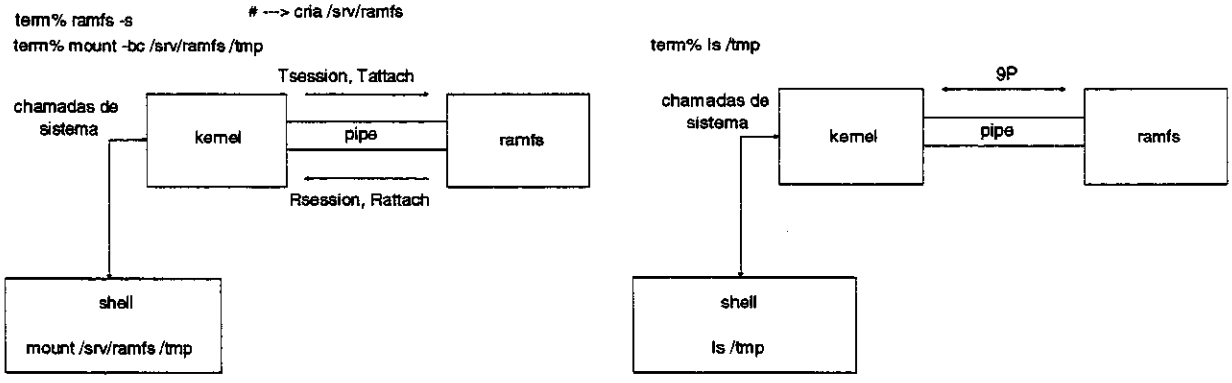
note — cadeias de caracteres escritas para o arquivo *note* serão marcadas como uma "nota" para o processo. O arquivo *notepg* é similar, mas a "nota" será entregue para todos os processos no *grupo de notas* do processo alvo. Entretanto, se o processo fazendo a escrita está no grupo, ele não receberá a nota. O arquivo *notepg* é somente para escrita.

noteid — Esse arquivo textual pode ser lido para recuperar um inteiro identificando o grupo de notas do processo. O arquivo pode ser escrito para fazer com que

o processo mude para um outro grupo de notas, desde que o grupo exista e seja de propriedade do mesmo usuário.

É importante esclarecer que os serviços que */proc* provê, embora variados, não excedem a noção de um processo como um arquivo. Por exemplo, não é possível terminar um processo tentando remover seu arquivo de processo, nem é possível iniciar um novo processo criando um arquivo de processo. Os arquivos dão uma visão ativa dos processos, mas eles não os representam literalmente. Esta distinção é importante quando se projeta serviços como sistemas de arquivos.

A Figura 3.8 ilustra um outro exemplo de serviço: o servidor *ramfs*. *Ramfs* é um servidor no nível do usuário que provê arquivos temporários em memória. Por sua simplicidade é também muito útil como um exemplo de como escrever um servidor de arquivos no nível do usuário. Ele inicia um processo que se monta no ponto de mount (default /tmp). O processo *ramfs* implementa uma árvore de arquivos com início ("raiz") em *dir*, mantendo todos os arquivos em memória. Inicialmente a árvore de arquivos está vazia.



(a) Extensão através de mount

(b) Uso de ramfs

Figura 3.8: Servidor no nível do usuário

3.5 Suporte do kernel para redes

O código para suporte a redes no Plan 9 é complexo. Implementações de protocolos consistem quase inteiramente de sincronização e gerenciamento dinâmico de memória, áreas demandando estratégias sutis de recuperação de erros. O kernel atualmente suporta Datakit, conexões de fibra ponto-a-ponto, o conjunto de protocolos Internet (IP) e

serviços de dados ISDN. A variedade de redes e máquinas suportadas pelo Plan 9 levantou questões de implementação não tratadas por outros sistemas operacionais executando sobre hardware comercial suportando somente Ethernet ou FDDI.

O código de rede no kernel está dividido em três camadas: interface de hardware, processamento de protocolo e interface de programa. Um driver de dispositivo tipicamente usa streams[1] para conectar as duas camadas de interfaces. Módulos adicionais de streams podem ser colocados em um dispositivo para protocolos do processo. Cada driver de dispositivo é um sistema de arquivos residente no kernel. Drivers de dispositivos simples servem um único diretório contendo alguns arquivos; por exemplo, representa-se cada UART por um arquivo de dados e outro de controle:

```
term % cd /dev
cpu% ls -l eia*
--rw-rw---- t 0 Franklin Franklin 0 Mar 16 17:50 eia0
--rw-rw---- t 0 Franklin Franklin 0 Mar 16 17:50 eia0ctl
--rw-rw---- t 0 Franklin Franklin 0 Mar 16 17:50 eia0
--rw-rw---- t 0 Franklin Franklin 0 Mar 16 17:50 eia0ctl
term%
```

O arquivo de controle é usado para controlar o dispositivo; escrevendo a cadeia *b1200* para */dev/eia1ctl* configura a linha para 1200 baud.

3.5.1 Dispositivos de protocolo

Conexões de rede são representadas como pseudo-dispositivos chamados "dispositivos de protocolo". Drivers de dispositivo de protocolo existem para o protocolo Datakit URP e para cada um dos protocolos Internet IP, TCP, UDP, e o protocolo do Plan 9, IL. Todos os dispositivos de protocolo parecem idênticos; desta forma programas do usuário não contêm código específico de rede.

Cada driver do dispositivo de protocolo serve uma estrutura de diretórios, como mostrado na Figura 3.9. O diretório do topo contém um arquivo *clone* e um diretório para cada conexão numerada de 0 a n. Cada diretório de conexão contém arquivos para controlar uma conexão e enviar e receber informações pela conexão. Um diretório de conexão TCP assemelha-se a este:

[1] streams são a camada de comunicação de mais baixo nível do Plan 9, normalmente invisíveis às aplicações.

```
term % cd /net/tcp/2
term% ls -l
--rw-rw---- I 0 Franklin Franklin 0 Mar 16 17:50 ctl
--rw-rw---- I 0 Franklin Franklin 0 Mar 16 17:50 data
--rw-rw---- I 0 Franklin Franklin 0 Mar 16 17:50 listen
--r--r--r-- I 0 Franklin Franklin 0 Mar 16 17:50 local
--r--r--r-- I 0 Franklin Franklin 0 Mar 16 17:50 remote
--r--r--r-- I 0 Franklin Franklin 0 Mar 16 17:50 status
term% cat local remote status
143.24.106.117!9002
143.106.24.84!5000
tcp/2 1 Established connect
term%
```

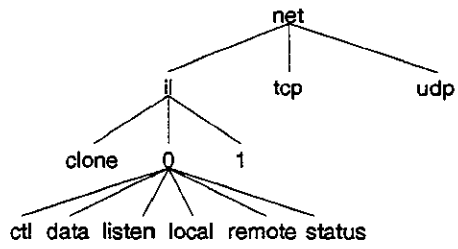


Figura 3.9: Árvore de diretórios dos dispositivos de protocolo

Os arquivos *local*, *remote*, e *status* fornecem informações sobre o estado da conexão. Os arquivos *data* e *ctl* provêm acesso ao processo do lado do stream implementando o protocolo. O arquivo *listen* é usado para aceitar chamadas chegando da rede. Os seguintes passos estabelecem uma conexão.

- O dispositivo *clone* do diretório do protocolo apropriado é aberto por *open* para reservar uma conexão não-usada.

- O descritor de arquivo retornado por *open* aponta para o arquivo *ctl* da nova conexão. A leitura do arquivo que esse descritor aponta retorna uma cadeia ASCII contendo o número de conexão.

- Uma cadeia ASCII definindo um protocolo/rede é escrita para o arquivo *ctl*. Por exemplo, para abrir uma sessão Telnet (porta 23) para uma máquina remota com endereço IP 143.106.24.117, a cadeia é:

  connect 143.106.24.117!23

- O caminho do arquivo *data* é construído usando o número da conexão. Quando o arquivo *data* é aberto a conexão é estabelecida.

Um processo pode ler e escrever este descritor de arquivo para enviar e receber mensagens da rede. Se o processo abre o arquivo *listen* ele bloqueia até que uma chamada chegando é recebida. Uma cadeia de caracteres é escrita no arquivo *ctl* antes que o listen selecione as portas ou serviços que o processo está preparado para aceitar. Por exemplo, para iniciar o listener Telnet remoto, a cadeia é:

announce 23

Quando uma chamada é recebida, *open* termina e retorna um descritor de arquivo apontando para o arquivo *ctl* da nova conexão. A leitura do arquivo *ctl* produz um número de conexão usado para construir o caminho do arquivo *data*. Uma conexão permanece estabelecida enquanto quaisquer dos arquivos no diretório de conexão estão sendo referenciados ou até que um *close* é recebido da rede.

3.5.2 "Banco de dados da rede"

Com o objetivo de facilitar a administração de redes, Plan 9 possui um banco de dados em um servidor compartilhado contendo todas as informações necessárias para tal administração, ao contrário das maioria dos sistemas que têm vários arquivos. Dois arquivos ASCII constituem o banco de dados principal: */lib/ndb/local* contém informações administradas localmente e */lib/ndb/global* contém informações importadas de qualquer outra localidade. Os arquivos contêm conjuntos de pares atributo/valor da forma *attr=valor*, onde *attr* e *valor* são cadeias alfanuméricas. Sistemas são descritos por tuplas de múltiplas linhas; uma linha de cabeçalho na margem esquerda inicia cada tupla seguida por zero ou mais pares identados atributo/valor especificando nomes, endereços, propriedades etc. Por exemplo, a entrada para o servidor de CPU da Bell-Labs especifica um nome de

domínio, um endereço IP, um endereço Ethernet, um endereço Datakit, um arquivo de boot, e protocolos suportados.

```
sys = helix
        dom=helix.research.att.com
        bootf=/mips/9power
        ip=135.104.9.31 ether=0800690222f0
        dk=nj/astro/helix
        proto=il flavor=9cpu
```

Se vários sistemas compartilham entradas tais como máscaras de rede e gateways, especifica-se essa informação na tupla rede ou subrede ao invés de na tupla sistema. As seguintes entradas definem uma rede IP Classe B e algumas subredes derivadas. A entrada correspondendo à rede especifica a máscara IP, sistema de arquivo, e servidor de autenticação para todos os sistemas na rede. Cada subrede especifica seu gateway IP default.

```
ipnet=mh-astro-net ip=135.104.0.0 ipmask=255.255.255.0
        fs=bootes.research.att.com
        auth=1127auth
ipnet=unix-room ip=135.104.117.0
        ipgw=135.104.117.1
ipnet=third-floor ip=135.104.51.0
        ipgw=135.104.51.1
ipnet=fourth-floor ip=135.104.52.0
        ipgw=135.104.51.1
```

Entradas do banco de dados também definem o mapeamento de nomes de serviços para números de portas TCP, UDP, e IL.

```
tcp=echo      port=7

tcp=discard   port=9

tcp=systat    port=11

tcp=daytime   port=13
```

Todos os programas lêem o banco de dados diretamente, logo, problemas de consistência são raros. Entretanto, os arquivos do banco de dados podem tornar-se grandes.

3.5.3 Servidor de conexão

Um importante serviço do sistema provido através de uma interface de arquivo é o servidor de conexão, um serviço que traduz o nome simbólico de um host para um endereço de rede, o qual pode ser usado para estabelecer uma conexão.

Um cliente escreve o nome do host de interesse no arquivo especial */net/cs*, e então lê de volta a resposta (Figura 3.10).

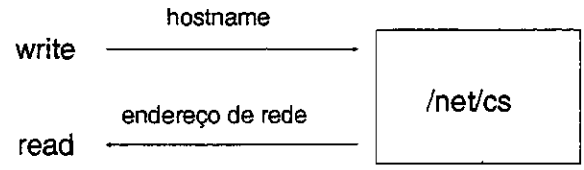


Figura 3.10: Servidor de conexão

CS usa informações sobre redes disponíveis, o banco de dados da rede, e outros servidores (tais como DNS) para traduzir nomes. O seguinte exemplo ilustra o uso de *cs*. *Ndb/csquery* é um programa que espera por cadeias de caracteres para escrever em */net/cs* e imprime as respostas.

```
%ndb/csquery
> net!helix!9fs
/net/il/clone 135.104.9.31!17008
/net/dk/clone nj/astro/helix!9fs
```

Para nomes de domínio *CS* primeiro consulta um outro processo à nível de usuário, o servidor de domínio de nomes (DNS). Se nenhum *DNS* é alcançável, *CS* baseia-se em suas próprias tabelas.

Como *CS*, o servidor de domínio de nomes é um processo a nível do usuário provendo um arquivo, */net/dns*. Um cliente escreve uma requisição da forma nome-domínio tipo, onde tipo é um recurso de serviço de domínio de nome que registra tipos. DNS executa uma consulta recursiva através do sistema de domínio de nomes Internet produzindo uma linha de registro por recurso encontrado. O cliente lê */net/dns* para recuperar os registros. Como outros servidores de domínios de nomes, *DNS* cacheia informações obtidas da rede. *DNS* é implementado como uma aplicação de memória compartilhada multiprocesso com processos separados escutando ("listening") requisições locais e de redes.

3.5.4 Rotinas de biblioteca

Rotinas de biblioteca são providas para liberar o programador dos detalhes de fazer e receber conexões através de uma rede.

Conectando

A chamada de biblioteca *dial* estabelece uma conexão a um destino remoto. Ela retorna um descritor de arquivo aberto para o arquivo *data* no diretório de conexão.

int **dial**(char *dest, char *local, char *dir, int *cfdp)

dest — é o nome/endereço simbólico do destino.

local — é o endereço local. Uma vez que a maior parte das redes não suporta isto, ele é usualmente zero.

dir — é um ponteiro para um buffer armazenar o nome do caminho do diretório do protocolo representando esta conexão. *Dial* preenche este buffer se o ponteiro é diferente de zero.

cfdp — é um ponteiro para um descritor do arquivo *ctl* da conexão. Se o ponteiro é diferente de zero, *dial* abre o arquivo de controle e coloca o descritor de arquivo nele.

A maioria dos programas chamam *dial* com um nome destino e todos os outros argumentos zerados. *Dial* usa *CS* para traduzir o nome simbólico para todos os endereços destino possíveis e tenta conectar a cada um deles até que um funcione. Especificando o nome especial *net* na porção da rede do destino permite *CS* escolher uma rede/protocolo em comum com o destino para qual o serviço requisitado é válido.

Dial aceita endereços ao invés de nomes simbólicos. Por exemplo, os destinos tcp!143.106 .24.117!513 e tcp!mercurio.dcc.unicamp.br!login são referências equivalentes à mesma máquina.

Listening

Um programa usa quatro rotinas para receber uma conexão. Ele primeiro executa *announce()* (para "anunciar") sua intenção de receber conexões, e então *listen()* ("para escutar") por chamadas e finalmente *accept()* ("aceita" a conexão) ou *reject()* ("rejeita" a conexão). *Announce* retorna um descritor de arquivo aberto pelo arquivo *ctl* de uma conexão e preenche *dir* com o caminho do diretório do protocolo para o anunciamento.

int **announce**(char *addr, char *dir)

addr é o nome/endereço simbólico anunciado; se ele não contém um serviço, o anunciamento é para todos os serviços não explicitamente anunciados. Conseqüentemente, um serviço pode facilmente escrever o equivalente do programa *inetd* sem ter tido que anunciar cada serviço em separado. Um anunciamento permanece em vigência até que o arquivo de controle seja fechado.

Listen retorna um descritor de arquivo aberto para o arquivo *ctl* e preenche *ldir* com o caminho do diretório do protocolo para a conexão recebida. Para ele é passado *dir* do anunciamento.

int **listen**(char *dir, char *ldir)

Accept e *reject* são invocados com o descritor do arquivo de controle e *ldir* retornados por *listen*. Algumas redes tais como Datakit aceitam uma razão para rejeição; redes tais como IP ignoram o terceiro argumento.

int **accept**(int ctl, char *ldir)
int **reject**(int ctl, char *ldir, char *reason)

O seguinte código implementa um "listener" TCP típico. Ele anuncia a si próprio, espera por conexões, e cria um novo processo para cada conexão. O novo processo ecoa dados na conexão até que o lado remoto o feche. O "*" no nome simbólico significa que o anunciamento é válido para qualquer endereço acoplado à máquina que o programa está executando.

```
int
echo_server(void)
{
  int dfd, lcfd;
  char adir[40], ldir[40];
  int n;
  char buf[256];

  afd = announce("tcp!*!echo", adir);
  if(afd < 0)
       return -1;
```

```
  for(;;){
        /* escuta por uma chamada*/
        lcfd = listen(adir, ldir);
        if(lcfd < 0)
              return -1;

        /* cria um processo para fazer o echo */
        switch(fork()){
        case 0:
              /* aceita a chamada e abre o arquivo data */
              dfd = accept(lcfd, ldir);
              if(dfd < 0)
                return -1;

              /* echo until EOF */
              while((n = read(dfd, buf, sizeof(buf))) > 0)
                    write(dfd, buf, n);
              exits(0);
        case -1:
               perror("forking");
        default:
               close(lcfd);
               break;
        }
  }
}
```

3.5.5 Comunicação entre máquinas Plan 9 e com outros sistemas

Além de *exportfs*, *import* e *cpu*, pode-se também usar *srv* e *9fs* para comunicação entre máquinas Plan 9. *Srv* é um servidor no nível do usuário que disca a máquina informada e inicializa a conexão para servir o protocolo 9P. Ele então cria em /srv um arquivo chamado *srvname*. Usuários podem então montar o serviço, tipicamente em um nome em */n*, para acessar os arquivos providos pela máquina remota. O serviço especificado deve servir 9P. *Srv* pode ser também usado para comunicação com algum sistema não-Plan 9. Nesse caso, um serviço como *u9fs* (visto na próxima seção) deve ser mencionado explicitamente.

O comando *9fs* é um script *rc* que faz o *srv* e o mount necessários para tornar

disponíveis os arquivos do sistema informado. Os arquivos são montados em um mount-point, se dado; caso contrário eles são montados em /n/system.

Como um exemplo de uso de *9fs* temos o uso do servidor *Ramfs* remotamente. Esta solução foi dada por Presotto através da lista 9fans e requer a criação do arquivo /rc/bin/service/il444 cujo conteúdo é:

```
#!bin/rc
ramfs -i
```

Com esse arquivo criado pode-se montar uma instância desse sistema de arquivos remotamente com

```
9fs il!<nome do sistema>!444
```

Sendo que cada chamada obtém uma nova instância de *ramfs*.

U9FS

Plan 9 possui um servidor (*u9fs*) que executa em sistemas UNIX e entende o protocolo 9P, de modo que sistemas de arquivos de máquinas UNIX podem ser importados para dentro do Plan 9, ou seja, *u9fs* é um programa executando numa máquina UNIX e que serve arquivos UNIX para máquinas Plan 9 usando o protocolo 9P. Ele deve ser invocado em uma máquina UNIX por *inetd* com sua entrada, saída e erro padrão conectados à conexão de rede, tipicamente TCP em Ethernet. Ele executa como usuário *root* e multiplexa acesso a múltiplos clientes Plan 9 através de uma única linha. Ele simula permissões UNIX supondo que identificadores de usuários (uids) do Plan 9 casam com nomes de login do UNIX.

U9fs foi usado extensivamente nos testes de desempenho envolvendo conexões do Plan 9 com os sistemas Linux e SunOS.

Plan 9 possui também um servidor compatível com NFS (*NFSServer*) que executa no Plan 9, de maneira que sistemas de arquivos Plan 9 possam ser acessados a partir de outros sistemas que suportem NFS. Plan 9 também possui o conjunto completo de protocolos Internet: *telnet*, *rlogin*, *ftp*.

Ftpfs

Não existe comando *ftp* no Plan 9. Ao invés disso, um servidor no nível de usuário chamado *ftpfs* disca o site FTP , loga nele em nome do usuário, e usa o protocolo FTP para examinar arquivos no diretório remoto. Ao usuário local, ele oferece uma hierarquia de arquivos, conectada ao espaço de nomes local no diretório */n/ftp*, refletindo os conteúdos do site

FTP. Em outras palavras, ele traduz o protocolo FTP em 9P para oferecer acesso Plan 9 aos sites FTP. *ftpfs* faz um cacheamento sofisticado maior para eficiência e usa heurísticas para decodificar informações de diretórios remotos. O resultado é significativo: todas as ferramentas de gerenciamento de arquivos locais tais como *cp*, *grep*, *diff*, e *ls* estão disponíveis para os arquivos servidos por FTP, exatamente como se eles fossem arquivos locais.

3.6 O protocolo IL

IL é um protocolo leve encapsulado por IP. Ele é baseado em conexão e provê transmissão confiável de mensagens seqüenciadas. Quando IL envia dados para o outro lado, ele requer um reconhecimento em retorno. Se um reconhecimento não é recebido, IL não retransmite automaticamente os dados, ou seja, ele não faz retransmissão às cegas. Primeiro ele envia uma mensagem perguntando ao servidor se ele recebeu a mensagem de dados enviada, em caso de resposta negativa do servidor, ele retransmite a mensagem de dados. Isto ajuda o desempenho em redes congestionadas, onde retransmissões às cegas podem causar congestionamentos futuros. Se após algum número de consultas, o servidor não responder, IL irá desistir.

Assim como TCP, IL contém algoritmos para estimar dinamicamente o *round-trip time* (RTT)("tempo de ida e volta") entre um cliente e um servidor, de maneira que ele sabe quanto tempo esperar por um reconhecimento. Por exemplo, o RTT em uma LAN pode ser milissegundos enquanto que através de uma WAN pode ser segundos. Além disso, IL pode medir um RTT de 1 segundo entre um cliente e um servidor e então 30 segundos mais tarde medir um RTT de 2 segundos na mesma conexão, causado por variações no tráfego da rede.

A configuração da conexão usa um handshake two-way ("protocolo bidirecional") para gerar números de seqüência iniciais em cada lado da conexão; mensagens de dados subseqüentes incrementam a seqüência de números para permitir o receptor reseqüenciar mensagens fora de ordem.

Nenhuma provisão é feita para controle de fluxo uma vez que o protocolo é projetado para transportar mensagens do tipo RPC entre cliente e servidor, que é uma estrutura com limitações inerentes de fluxo. Uma pequena janela para mensagens recebidas pendentes previne muitas mensagens de serem buferizadas; mensagens fora da janela são descartadas e devem ser retransmitidas.

3.6.1 Estabelecimento e término de uma conexão IL

Para um melhor entendimento das funções *dial*, *announce*, *listen* e *accept*, apresentamos a seguir como conexões IL são estabelecidas e terminadas.

Conexões não usadas estão no estado *Closed* sem endereços ou portas atribuídos. Dois eventos abrem uma conexão: a recepção de uma mensagem cujo endereço e porta casam com alguma conexão não aberta ou um usuário abrindo explicitamente uma conexão. No primeiro caso, o endereço e porta da fonte da mensagem tornam-se o endereço e porta remotos da conexão e o endereço e porta do destino da mensagem tornam-se o endereço e porta local. Isto representa que o servidor fez um *listen* em uma porta. O estado da conexão é ligado para *Syncee* e a mensagem é processada. No segundo caso, o usuário especifica tanto os endereços e portas locais quanto remotos. Isto corresponde ao cliente fazendo *dial* para o servidor. O estado da conexão é ajustado para *Syncer* e uma mensagem *sync* é enviada para o lado remoto. Os valores legais para os endereços locais estão restritos pela implementação IP.

O número mínimo de pacotes requeridos para esta troca é dois; conseqüentemente isto é chamado de *two-way handshake* do IL. A Figura 3.11 mostra os dois segmentos e a seguir os resultados das execuções do comando *netstat* no cliente e no servidor.

No servidor, após fazer o listen na porta 9002:

il 0 Franklin Listening 9002 0 0.0.0.0 il

No servidor, após começar a receber as mensagens do cliente:

il 1 Franklin Established 9002 5000 143.106.24.84

No cliente, após o início do envio de mensagens:

il 0 robert Established 5000 9002 143.106.24.117

No servidor, após o término do envio de mensagens:

il 0 robert Closed 5000 9002 143.106.24.117

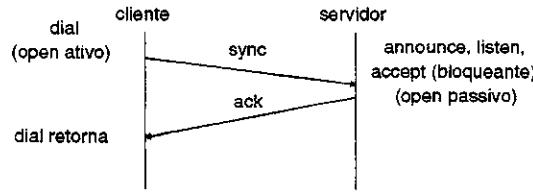


Figura 3.11: IL two-way handshake

As semânticas do *close* do IL são diferentes do TCP. Se o remetente fecha a conexão, o destinatário descarta quaisquer mensagens ainda enfileiradas no kernel, o que gerou considerável confusão nos testes, até o problema ser esclarecido.

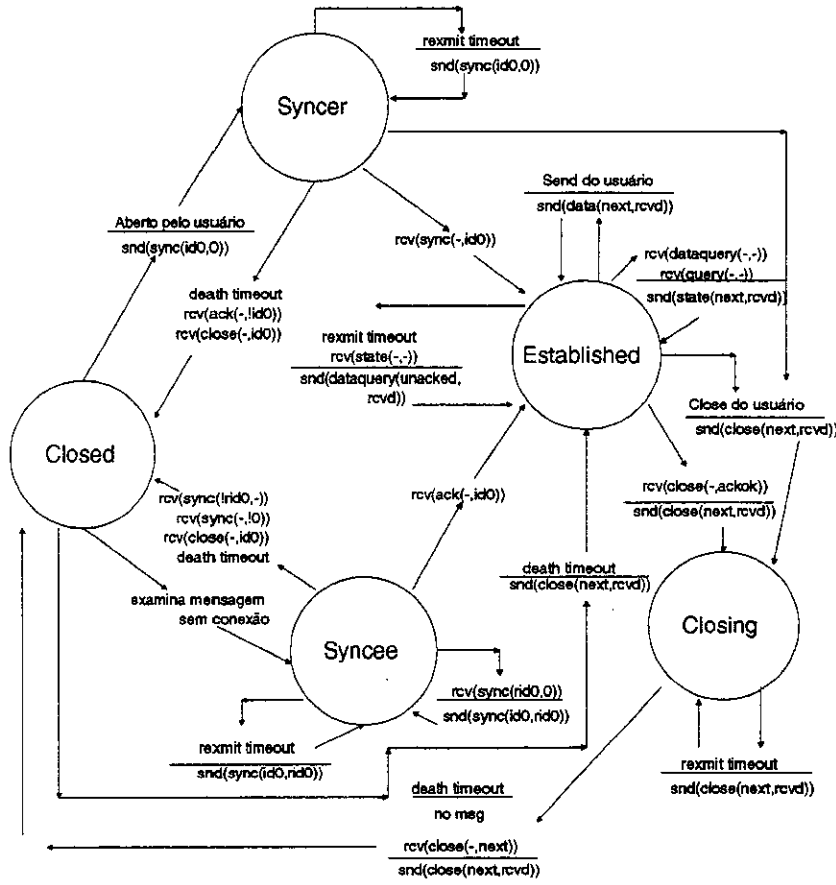
3.6.2 Diagrama de Transição de Estados

Uma conexão IL transporta um stream de dados entre dois pontos. Enquanto a conexão persiste, dados entrando de um lado são enviados para o outro lado na mesma seqüência. O funcionamento de uma conexão é descrito pela máquina de estados da Figura 3.12 usando a notação comumente empregada nesse tipo de representação gráfica. Essa figura mostra os estados (círculos) e transições entre estados (arcos). Cada transição é rotulada com a lista de eventos que podem causar a transição e, separados por uma linha horizontal, as mensagens enviadas ou recebidas nessa transição.

A máquina de estados IL tem cinco estados: *Closed*, *Syncer*, *Syncee*, *Established*, e *Closing*. A conexão é identificada pelo endereço IP e número da porta usados em cada lado. Os endereços estão contidos no cabeçalho do protocolo IP, enquanto as portas são partes do cabeçalho de 18 bytes do IL. As variáveis locais identificando o estado de uma conexão são:

state	um dos estados
laddr	endereço IP local de 32 bits
lport	porta IL local de 16 bits
raddr	endereço IP remoto de 32 bits
rport	porta IL remota de 16 bits
id0	número de seqüência inicial de 32 bits do lado local
rid0	número de seqüência inicial de 32 bits do lado remoto
next	número da seqüência da próxima mensagem a ser enviada do lado local

rcvd	a última mensagem em seqüência recebida pelo lado remoto
unacked	número da seqüência da primeira mensagem unacked



ackok	qualquer número de seqüência entre *id0* e *next* inclusive
!x	qualquer valor exceto x
-	qualquer valor

Figura 3.12: Transições de estado do IL

3.6.3 Estrutura do cabeçalho

A seguir tem-se uma descrição em linguagem C de um cabeçalho IP+IL, não considerando opções IP:

```
typedef unsigned char byte
struct IPIL
{
byte vihl;      /* Versão e tamanho do cabeçalho */
byte tos;       /* Tipo do serviço */
byte length[2]; /* Tamanho do pacote */
byte id[2];     /* Identificação */
byte frag[2];   /* Fragmento de informação */
byte ttl;       /* Time to live */
byte proto;     /* Protocolo */
byte cksum[2];  /* Checksum do cabeçalho */
byte src[4];    /* IP do Fonte */
byte dst[4];    /* IP do Destino */
byte ilsum[2];  /* Checksum incluindo cabeçalho */
byte illen[2];  /* Tamanho do pacote */
byte iltype;    /* Tipo do pacote */
byte ilspec;    /* Especial */
byte ilsrc[2];  /* Porta do fonte */
byte ildst[2];  /* Porta do destino */
byte ilid[4];   /* Id da seqüência */
byte ilack[4];  /* Seqüência acked */
}
```

Dados devem seguir imediatamente após o cabeçalho na mensagem. *ilspec* é uma extensão reservada para mudanças futuras no protocolo.

O checksum é calculado com *ilsum* e *ilspec* ajustados para zero. Ele é o checksum padrão IP, isto é, 16 bits do complemento de um da soma de todas as palavras de 16 bits no cabeçalho e texto. Se uma mensagem contém um número ímpar de bytes no cabeçalho e no texto, o último byte é preenchido à direita com zeros para formar uma palavra de 16 bits para o checksum.

Os valores possíveis de iltype são:

sync	=	0
data	=	1
dataquery	=	2
ack	=	3
query	=	4
state	=	5
close	=	6

O campo illen é o tamanho em bytes do cabeçalho IL (18 bytes) mais o tamanho dos dados.

IL transporta mensagens de dados. Cada mensagem corresponde a uma única escrita do sistema operacional e é identificada por um número de seqüência de 32 bits. O número de seqüência inicial para cada direção em uma conexão é escolhido aleatoriamente e transmitido na mensagem inicial *sync*. O número é incrementado para cada mensagem de dados subseqüente. Uma mensagem retransmitida contém seu número de seqüência original.

3.6.4 Temporizadores

Cada mensagem contém dois números de seqüência: um identificador (ID) e um reconhecimento ("ack"). O reconhecimento é o último na seqüência de dados recebida pelo transmissor da mensagem. Para mensagens *data* e *dataquery*, o ID é seu número de seqüência. Para mensagens de controle *sync*, *ack*, *query*, *state*, e *close*, o ID é um número maior do que o maior número de seqüência da mensagem do tipo "data" enviada.

O remetente transmite mensagens de dados com o tipo *data*. Quaisquer mensagens trafegando na direção oposta transportam reconhecimentos. Uma mensagem "ack" será enviada dentro de 200 milissegundos após o recebimento da mensagem de dados a menos que uma mensagem retornando já tenha um reconhecimento "piggy-backed" para o remetente.

Em IP, mensagens podem ser entregues fora de ordem ou podem ser perdidas devido a congestionamento ou falhas. Para superar isto, IL usa um protocolo "go back n" modificado, que tenta evitar agravar o congestionamento da rede. Um "round-trip time" médio é mantido medindo o atraso entre a transmissão de uma mensagem e a recepção de seu reconhecimento. Até que o primeiro reconhecimento seja recebido, o "round-trip time" médio é de 100ms. Se um reconhecimento não é recebido dentro de quatro "round-trip times" da primeira mensagem sem reconhecimento ("rexmit timeout" na Figura 3.12), IL supõe que a mensagem ou o reconhecimento foram perdidos. O remetente então reenvia somente a primeira mensagem sem reconhecimento, ajustando o tipo para *dataquery*. Quando o destinatário recebe uma mensagem *dataquery*, ele responde com uma men-

sagem *state* reconhecendo a mensagem de dados de mais alta seqüência recebida. Esta mensagem pode ser a mensagem retransmitida ou, se o recebedor salvou mensagens fora de seqüência, alguma mensagem com número mais alto. Implementações do destinatário são livres para escolher se salvam mensagens fora de ordem. Quando o remetente recebe a mensagem *state*, ele irá imediatamente reenviar a próxima mensagem não reconhecida com tipo *dataquery*. Isto continua até que todas as mensagens sejam reconhecidas.

Se nenhum reconhecimento é recebido após a primeira mensagem *dataquery*, o transmissor continua a temporizar e reenviar a mensagem *dataquery*. Os intervalos entre retransmissão crescem exponencialmente. Após 300 vezes o "round trip time" ("death timeout" na Figura 3.12), o remetente desiste e supõe que a conexão está morta.

Retransmissão também ocorre nos estados *Syncer*, *Syncee*, e *Close*. Os intervalos de retransmissão são os mesmos para mensagens de dados.

Conexões a sistemas mortos devem ser descobertas e finalizadas a fim de que elas não consumam recursos. Se o sistema sobrevivente não precisa enviar quaisquer dados e todos os dados que ele enviou foram reconhecidos, o protocolo descrito até aqui não descobrirá essas conexões. Conseqüentemente, no estado *Established*, se outras mensagens não são enviadas por um período de 6 segundos, uma mensagem *query* é enviada. O destinatário sempre responde a uma mensagem *query* com uma mensagem *state*. Se mensagens não são recebidas por 30 segundos, a conexão é finalizada. Isto não está mostrado na Figura 3.12.

Capítulo 4

Instalação do Plan 9 em ambiente PC

Este capítulo tenta ser um pequeno manual de instalação, configuração e uso do sistema Plan 9 na plataforma PC, mostrando os procedimentos a serem executados durante essas três fases. Com a experiência adquirida na instalação do Plan 9 em 6 máquinas PCs, sendo 4 máquinas Pentium e 2 486, além de várias reinstalações e tentativas de reinstalações em tais máquinas, apresentaremos algumas pistas e alguns problemas ocorridos durante essas fases. Também apresentamos alguns endereços para documentos bastante úteis.

4.1 Distribuição

Plan 9 é distribuído de duas formas: um conjunto com quatro disquetes, contendo os binários para a instalação do sistema na plataforma PC, que está disponível para download gratuitamente na Internet em *http://plan9.bell-labs.com/plan9/distribution.html* e um kit de instalação contendo os quatro disquetes referidos anteriormente, um CD com todos os fontes e binários para as plataformas nas quais Plan 9 executa, e dois manuais: "The Documents", contendo vários papers sobre o sistema, e "The Manuals", um manual de uso dos comandos, ferramentas, servidores, configuração etc.

É importante ressaltar que os manuais estão também disponíveis no site da Bell-labs na versão HTML, sendo que existe uma ferramenta para busca on-line no manual "The Manuals". Esta mesma ferramenta está disponível no sistema Plan 9 após a instalação.

Nossa experiência com Plan 9 começou com o download do sistema em quatro disquetes e posterior instalação e configuração em rede do mesmo em 2 máquinas. Após essa fase adquirimos o kit de instalação do sistema completo.

4.2 Instalação

Para a instalação do sistema é necessário uma partição DOS de 4MB para instalação dos arquivos de configuração e inicialização do sistema, incluindo-se aqui o kernel. Além disso é necessário ter-se um espàço livre de pelo menos 20MB livres no final do HD para a instalação da versão com quatro disquetes. Para a instalação completa via CD é necessário um espaço no final do disco de 600MB. Estes espaços não devem ser uma partição. Devem ser realmente um espaço livre e têm que estar no final do disco. Como exemplo, citamos a nossa configuração de um HD IDE de 3093MB no qual Plan 9 está instalado.

/dev/hda1	DOS	32MB
/dev/hda2	OS/2 HPFS	900MB
/dev/hda3	Linux swap	64MB
/dev/hda4	Linux native	1GB

Como Plan 9 não busca informações na BIOS, é necessário, durante a instalação do primeiro disquete, informar a configuração dos dispositivos instalados, tais como placa de rede, mouse, CD-ROM, placa de som, monitor. Talvez não se encontre a configuração para a placa de rede, devendo-se então escolher as opções padrão referentes a essa configuração (I/O Base and Interrupt) e, quando o sistema estiver instalado deve-se informar os parâmetros corretos. Esta fase é válida para os dois tipos de instalações: via quatro disquetes e CD-ROM, uma vez que o primeiro disquete é usado para as duas fases. Essas informações são gravadas no arquivo plan9.ini. Em seguida o sistema é reinicializado.

A instalação do primeiro disquete cria um diretório chamado plan9, o qual contém vários arquivos, tais como: plan9.ini, o kernel do Plan 9 etc. Deve-se então ir para esse diretório (c:plan9 ou d:plan9, dependendo do nome da partição onde o sistema está instalado) e digitar o comando "b.com" ou simplesmente "b" que é um carregador do kernel do Plan 9 a partir do DOS. Em seguida surge uma tela em que o usuário pode optar pela instalação do conjunto de quatro disquetes (agora só serão três, pois um já foi instalado) ou do CD-ROM. Durante essa fase pode-se optar também pela instalação de um servidor de CPU ou servidor de arquivos. O sistema será então instalado no espaço livre no final do disco.

Após copiar todo o conteúdo do CD para o disco, surge uma tela perguntando se o usuário deseja tornar o sistema ativo, confirma-se e surge uma mensagem de erro - um bug do sistema. Deve-se reiniciar e ir para o diretório plan9, onde deve-se editar o arquivo "plan9.ini", o qual contém todas as informações de configuração do sistema informados na primeira fase da instalação, substituindo a linha *bootdisk=#H/hd0disk:2096640* por *bootfile=h!0*. Essa linha fará com que o kernel seja carregado da partição plan 9 (a do final do disco). A menos que o usuário mude de kernel, o que vai ser carregado é o 9dos. Se

a placa de rede não estiver configurada com os parâmetros corretos, deve-se configurá-la nesse momento também nesse arquivo, alterando-se os parâmetros default escolhidos pelos parâmetros corretos. Salva-se as informações e digita-se no prompt do sistema "b.com". Este comando faz agora com que o kernel do Plan 9 seja carregado. Abaixo listamos o arquivo plan9.ini de uma de nossas máquinas.

```
ether0=type=NE2000 port=0x6100 irq=09
mouseport=0
console=cga
monitor=multisync75
vgasize=640x480x1
ata1=irq=15
#bootfile=h!0
#Estou usando 9dos devido a alteração de SEGMAPSIZE 64
#no arquivo /sys/src/9/pc/mem.h
bootfile=hd!0!/plan9/9pcdisk
rootdir=/plan9
```

Nota-se que a linha bootfile=h!0 está comentada, isso deve-se ao fato de termos gerado um novo kernel (9pcdisk) e o termos copiado para o diretório *c:/plan9/*. Portanto, a linha *bootfile=hd!0!/plan9/9pcdisk* informa ao carregador do sistema (b.com) que o kernel a ser utilizado é o 9pcdisk.

Na tela de inicialização do sistema, vê-se o endereço ethernet da placa de rede, o qual deve ser anotado para uso posterior. Nesse momento o sistema de inicialização requisitará o servidor de arquivos que será usado e o método através do qual o sistema se conectará a ele.

```
root is from (local, 9600, 19200, il)[local!#H/hd0fs]:
```

Se o terminal for uma máquina standalone deve-se pressionar Enter.

Os métodos de boot podem ser:

cyc — conecta-se via uma conexão ponto-a-ponto de fibra usando placas Cyclone. Se especificado, o endereço deve ser o número da placa Cyclone a ser usada. O valor default é 0.

il — conecta-se via Ethernet usando o protocolo IL.

tcp — conecta-se via Ethernet usando o protocolo TCP. Este método é usado somente se o servidor de arquivos inicial está em um sistema UNIX.

hs	conecta-se via Datakit usando o cartão Datakit de alta-velocidade.
incon	conecta-se via Datakit usando a interface Incon.
9600	conecta-se via Datakit usando a interface serial em 9600 baud.
19200	conecta-se via Datakit usando a interface serial em 19200 baud.
local	conecta-se ao sistema de arquivos local. Este foi o método que usamos.

É solicitado um nome de usuário, o qual deve ser *none* se a instalação foi feita a partir do conjunto de quatro disquetes e *tor* para a instalação feita a partir do CD-ROM. Quando a senha ("password") for requerida, deve-se apenas digitar <Enter>.

O primeiro recurso que deve-se conhecer sobre o novo ambiente Plan 9 instalado é o sistema de arquivos local. Quando se está executando um sistema de arquivos Plan 9 do HD, um programa chamado *kfs* provê o serviço de arquivos. *Kfs* é um servidor de arquivos local no nível de usuário para um terminal Plan 9 com um disco, que no nosso caso é um PC comum. Ele mantém um sistema de arquivos hierárquico no disco e oferece acesso 9P a ele. *Kfs* tem um conjunto de comandos para configurar e controlar o sistema de arquivos. Para a criação de um novo usuário deve-se executar os seguintes comandos:

disk/kfscmd allow

para permitir que qualquer usuário leia e escreva nos arquivos do sistema. Sendo que *dissallow* restaura a verificação normal de permissões.

/sys/lib/kfsuser "nome-do-usuário",

criará o usuário "nome-do-usuário" e estabelecerá um ambiente básico Plan 9 para esse usuário.

disk/kfscmd user

fará com que *kfs* registre o usuário no arquivo /adm/users

Com relação a password deve-se apenas pressionar Enter até que se tenha um servidor de autenticação funcionando.

Antes de reiniciar o sistema deve-se executar:

disk/kfscmd halt

para que todos os arquivos sejam fechados.

Durante a inicialização do sistema é muito comum ver a seguinte mensagem:

controller not in /lib/vgadb

Isto deve-se ao fato do Plan 9 (versão CD-ROM) reconhecer poucas placas de vídeo. O que se deve fazer então é atualizar os arquivos referentes à placa. Geralmente os seguintes procedimentos resolvem esse problema:

copiar os arquivos vga e vgadb de ftp.plan9.com/plan9/update/cmd/aux/vga.

copiar o arquivo vga para /386/bin/aux/vga e vgadb para /lib/vgadb. Executar o seguinte comando:

term%**chmod** +x vga

Reiniciar o sistema.

Em nossas máquinas usamos os seguinte parâmetros ("plan9.ini"):

vgasize=1024x768x8

monitor=multisync135

As informações acima são concernentes à instalação de um terminal Plan 9 em um HD IDE. Em um HD SCSI os procedimentos são basicamente os mesmos, tendo apenas que informar ao sistema que o HD é do tipo SCSI.

4.2.1 Configuração da rede

Como visto no capítulo 3, Plan 9 mantém todas as informações referentes a redes no arquivo */lib/ndb/local*. Uma vez que a placa de rede foi reconhecida, deve-se proceder à configuração desse arquivo. Os parâmetros que devem ser informados são (para o nosso ambiente computacional):

```
#external internet domain service
dom=
ns=dcc.unicamp.br
dom=dcc.unicamp.br ip=143.106.7.8
#Seu sistema
sys = mercurio
```

```
dom=mercurio.dcc.unicamp.br
ip=143.106.24.117 ether=0000214bb1dd
bootfile = /386/9pcdisk
proto=tcp
proto=il
#endereços ip das redes e subredes
ipnet=ic-unicamp-net ip=143.106.0.0 ipmask=255.255.255.192 ipgw=143.106.24.65
```

O arquivo */lib/ndb/local* já existe contendo informações da rede da Bell-labs, deve-se então somente alterá-lo com as informações acima e comentar ou remover as demais linhas, deixando sem alterações somente as configurações de serviço: TCP, IL e UDP. Com esse arquivo configurado, o sistema está em rede. Nesse momento, pode-se também inicializar o servidor de nomes.

Como dito antes, o sistema já possui um arquivo */lib/ndb/local*, bastando apenas modificá-lo para que se adeque a sua rede. Vários atributos importantes são:

sys	nome do sistema
dom	nome de domínio Internet
ip	endereço Internet
ether	endereço Ethernet
dk	endereço Datakit
bootf	arquivo para download para executar o boot e carregar o sistema operacional ("bootstrap")
ipnet	nome da rede Internet
ipmask	máscara da rede Internet
ipgw	gateway Internet
auth	servidor de autenticação a ser usado
fs	servidor de arquivos a ser usado
tcp	um nome de serviço TCP
udp	um nome de serviço UDP
il	um nome de serviço IL

port	um número de porta TCP, UDP ou IL
restricted	um serviço TCP que pode ser chamado somente por portas com números menores que 1024
proto	um protocolo suportado por um host. O par *proto=il* é requerido pelo *cs* em tuplas para hosts que suportam o protocolo IL.
9P	parâmetros para o protocolo de arquivos 9P, em particular se o servidor autentica (9P=auth).

O arquivo */lib/ndb/auth* é usado durante autenticação para decidir quem tem poder para falar por (relação "speaks for") usuários.

Para que uma máquina Plan 9 acesse outra máquina Plan 9, é preciso executar os seguintes comandos na máquina remota que vai ser acessada:

term%**aux/listen** tcp il
term%**aux/listen** -t /bin/service

Esses comandos iniciam um listener para os serviços providos por Plan 9, tais como *exportfs*. Pode-se então, usar comandos de comunicação em rede, tais como *import*, *telnet*, *ftpfs*.

É importante ressaltar que quando se acessa um sistema Plan 9 deve-se, em princípio, usar o servidor de autenticação, o qual não está instalado ainda. Por isso deve-se usar o usuário *none* nos programas *telnet*, *ftpfs*, e dissabilitar a parte que invoca o servidor de autenticação nos códigos fontes dos programas tais como *import*, *exportfs*.

4.2.2 Instalação de um servidor de CPU

De acordo com o manual do Plan 9, é necessário um servidor de arquivos para a instalação de um servidor de CPU. Mas, o servidor de arquivos não é imprescindível.

Pode-se instalar um servidor de CPU primeiro instalando um terminal standalone e então inicializando-o com um kernel de CPU (usamos o 9PCCDPUDISK). Se a instalação default não cria um partição NVRAM para ele, tira-se 512 bytes da partição de boot.

De um terminal, pode-se executar o seguinte comando:

cpu -h pinduca

onde pinduca é o nome de nosso servidor de CPU.

Isto fará com que o espaço de nomes do terminal seja modificado de várias maneiras: o espaço de nomes do terminal agora é o espaço de nomes do servidor de CPU, mas o

espaço de nomes do terminal está em */mnt/term*; além disso, *bind* foi usado para colocar parte desse espaço de nomes antes do espaço de nomes do CPU Server. Ele também tenta colocar o usuário de volta no diretório onde estava no momento da execução do comando *cpu*. O protocolo de transporte sendo utilizado é o IL.

Pode-se invocar $8\frac{1}{2}$ nessa janela, então criar uma janela no gerenciador de janelas resultante.

Como não tínhamos um servidor de autenticação tivemos que comentar a função que o invocava em */sys/src/9/boot/key.c*.

Como nosso servidor de CPU tem dois HDs e o Plan 9 está instalado no segundo HD (para o Plan 9 o primeiro HD é chamado hd0 e o segundo hd1), tivemos que alterar o arquivo */sys/src/9/pc/pccpudisk*. Na linha que tinha boot cpu # H/hd0 substituímos por boot cpu #H/hd1.

4.2.3 Administração

Abaixo, mostramos os diretórios mais usados na estrutura do espaço de nomes convencional do sistema Plan 9.

/usr	Diretório de usuários do sistema.
/usr/xx/lib	Diretório que contém o perfil do usuário xx.
/usr/xx/bin	Diretório que contém os binários do usuário xx.
/sys/src/cmd	Fontes dos comandos do sistema.
/sys/src/ape/lib	Fonte de funções do UNIX implementadas sob Plan 9. Em C e Assembly.
/rc/bin/service	Scripts dos serviços de rede.
/dev	Diretório de dispositivos do sistema.
/sys/src/games	Fontes dos jogos que vêm com o sistema.
/sys/man	Manual do sistema.
sys/doc	Alguns papers antigos sobre o sistema.
net	Dispositivos de rede.
srv	Armazena conexões aos servidores de arquivos.

n Um diretório contendo pontos de mount para árvores de arquivos importadas de sistemas remotos.

tmp Um diretório de arquivos temporários do usuário.

O acesso ao HD (outras partições DOS diferentes da partição Plan 9) e disquete se dá através do servidor *dossrv*. O acesso a unidade de CD-ROM é feita através do servidor *9660srv*. Existem arquivos de scripts prontos para tais acessos. Os scripts *a:*, *c:*, *d:* acessam o floppy, o HD, e a unidade de CD-ROM respectivamente.

4.2.4 Kernel

Embora o kernel dependa criticamente das propriedades do hardware subjacente, a maior parte das funções de alto-nível do kernel, incluindo gerenciamento de processos, pseudo-dispositivos, e alguns códigos de rede, são independentes da arquitetura do processador. O código portável do kernel está dividido em duas partes: aquela implementando funções do kernel e aquela dedicada ao processo de boot. O código da primeira parte é armazenado no diretório */sys/src/9/port* e o código de boot é armazenado em */sys/src/9/boot*. O código do kernel de cada arquitetura é armazenado nos subdiretórios de */sys/src/9* nomeados para cada arquitetura. Logo, temos:

/sys/src/9/port implementa funções do kernel.

/sys/src/9/boot funções para o processo de boot.

/sys/src/9/arquitetura código do kernel dependente da arquitetura. No nosso caso a arquitetura é igual a 386.

O código portável do boot é compilado em uma biblioteca para cada arquitetura. Um programa principal específico de arquitetura é carregado com a biblioteca apropriada e o executável resultante é compilado dentro do kernel onde ele é executado como um processo de usuário nos estágios finais de inicialização do kernel. O processo de boot executa autenticação, liga o espaço de nomes raiz ao sistema de arquivos apropriado e inicia o processo *init*.

A organização do código fonte portável do kernel difere daquele da maior parte de código específico de outras arquiteturas. Ao invés de armazenar o código portável em uma biblioteca dedicada à arquitetura e carregar o código específico da arquitetura com essa biblioteca, o código portável é compilado no diretório associado com a arquitetura e ligado com os arquivos objetos específicos da arquitetura lá armazenados.

O kernel do Plan 9 reside em */dev/hd0fs*. Logo, quando a linha *h!0* é lida do arquivo plan9.ini é carregado o kernel que está em hd0fs. Pode-se também carregar o kernel a

partir da linha de comando, como por exemplo: *c: plan9>b.com hd!0!/plan9/9dos*. Isso pode ser feito de um disquete também.

Pode-se ainda copiar o kernel para hd0fs, não sendo mais necessário a alteração no arquivo plan9.ini.

4.2.5 Documentação

Um documento importante é CHANGES.TXT que descreve várias atualizações de arquivos do Plan 9, assim como o local onde encontrar tais arquivos atualizados. Este documento está em *ftp://plan9.bell-labs.com/plan9/update/*.

Um outro documento que dá várias dicas sobre a utilização e configuração do sistema é "Plan 9 Tips and Information". Este documento está em *http://www.ecf.toronto.edu/plan9/info/*.

Uma outra boa fonte de informações e repositório de arquivos de atualização é o site do Charles Forsyth: *http://www.caldo.demon.co.uk/plan9/*.

Além dos manuais e dos documentos citados acima, uma outro boa fonte de informação é a lista de discussão da comunidade de usuários Plan 9: 9fanscse.psu.edu. Esta lista não têm muitos usuários, mas a maioria dos seus integrantes são pessoas que têm um grande conhecimento acerca do Plan 9. Dentre essas pessoas está a equipe de desenvolvimento do Plan 9. Como Plan 9 é na sua maioria usado academicamente, os usuários nessa lista, em geral são pessoas do mundo acadêmico: professores e estudantes de pós-graduação. Além de nós, não existem mais brasileiros participando dessa lista, o que nos leva a concluir que somos os únicos a trabalhar com o Plan 9 no Brasil. É importante ressaltar que a maioria desses usuários têm sistemas Plan 9 completos, ou seja, com terminais, servidor de CPU, servidor de arquivos funcionando perfeitamente. Achamos que um sistema grande, além do da Bell-labs, é o da University of York, Inglaterra.

Por já participarmos da lista há algum tempo, às vezes fizemos perguntas diretamente para algum membro. Em particular sempre consultamos David Butler(db Systems), Russ Cox (Bell-Labs), Charles Forsyth (PhD University of York e Vita Nuova, Limited - uma software house do sistema Inferno) e Dave Presotto (Bell-Labs), os quais sempre nos responderam prontamente.

4.3 SPARC

Seguimos os procedimentos do Manual do Plan 9 para a instalação do servidor *u9fs*. Mas para que u9fs.c fosse compilado, foi preciso comentar as seguinte funções no arquivo libc.h:

memcmp, *strcmp*, *strcpy*, *strlen* e *sys_errlist*

4.4 Problemas encontrados

Plan 9 tem alta sensibilidade ao hardware sendo usado. Isso nos consumiu muito tempo em testes de vários dispositivos, tais como a unidade de CD-ROM ATAPI, essencial para instalação do sistema na versão em CD. Todos as nossas unidades são ATAPI, mas a única que funcionou foi um CD-ROM IBM 6x. Descobrimos isso testando cada unidade por vez, e somente essa funcionou. Após a instalação o sistema voltou a não reconhecer a unidade de CD, o que nos levou a ter que obter uma nova versão do drive de CD e adaptá-lo ao sistema, o que exigiu muito conhecimento acerca do sistema. Também tivemos problemas com algumas placas de rede padrão ISA, e no final descobrimos que Plan 9 só reconhece placas de rede padrão PCI.

Os micro-benchmarks de comunicação usam sockets e como Plan 9 implementa sockets, tentamos usar os micro-benchmarks sem modificação, mas não funcionou. Existem alguns problemas na implementação de sockets do Plan 9, talvez por que nenhum usuário do sistema usa tais funções. Optamos por usar as próprias funções de comunicação nativas do Plan 9 que funcionam e são bem mais simples que sockets.

A partir da versão em CD-ROM é possível instalar um Servidor de Arquivos Plan 9 na arquitetura PC usando um HD SCSI. Tentamos tal instalação, mas não conseguimos, pois o servidor morre na hora do boot. Examinamos o código fonte e pedimos ajuda para os usuários da lista, mas no final não conseguimos instalar o Servidor de Arquivos.

Algumas vezes, durante o uso dos protocolos de comunicação (IL, TCP e UDP) entre processos na mesma máquina ou entre máquinas diferentes, o sistema travava ou surgia algum lixo na tela. Esse problema foi resolvido aplicando-se os arquivos de atualização encontrados no documento CHANGES.TXT e no site do Forsyth no sistema.

A documentação não ajuda muito, sendo que o manual do sistema fornece informações de forma resumida, o que, às vezes, leva a um não entendimento do assunto pesquisado, o que nos levou a buscar a informação diretamente do código fonte do sistema, o que demandou mais tempo, mas ao mesmo tempo, trouxe mais conhecimento e experiência de uso do sistema.

Capítulo 5

Medidas de desempenho

5.1 Introdução a benchmarks de sistemas operacionais

Benchmarks podem ser categorizados de dois modos. Um modo é categorizar um benchmark como sendo um benchmark sintético ou de aplicação; o outro modo é como um macro ou micro-benchmark.

Benchmarks de aplicação consistem de programas e utilitários que um usuário pode de fato usar. Por exemplo, SPECint92 consiste de seis aplicações incluindo um compilador C, um interpretador lisp, e uma planilha eletrônica. Benchmarks sintéticos, de outro lado, modelam uma carga de trabalho (workload) através da execução de várias operações em uma combinação consistente com o workload alvo; os benchmarks do NFS (nfsstone, nhfsstone e LADDIS) permitem que o usuário entre uma combinação alvo de operações, isto é, que percentagem do workload deverá consistir de *creates*, *getattrs*, etc.

Benchmarks sintéticos são mais flexíveis do que benchmarks de aplicação: eles usualmente têm um grande número de parâmetros que permitem uma melhor adequação de acordo com a tecnologia e permitem um aumento no número de workloads diferentes que eles podem modelar. Entretanto, o problema com benchmarks sintéticos é que eles não medem qualquer trabalho real. Isto torna seus resultados questionáveis pois as operações completadas em um benchmark sintético podem não levar a mesma quantidade de tempo em uma aplicação real. Tanto o benchmark sintético pode adicionar sobrecarga ("overhead") que não existe em uma aplicação real, quanto uma aplicação real pode incorrer sobrecarga não modelada no benchmark.

A segunda maneira para categorizar um benchmark é como um macro-benchmark ou um micro-benchmark. Macro-benchmarks dão uma boa idéia da velocidade do sistema na sua totalidade, mas não provêm muita informação sobre que plataformas são melhores que outras. Eles usualmente modelam algum workload e podem ser tanto benchmarks sintéticos quanto de aplicação.

Micro-benchmarks medem características específicas do hardware ou sistema operacional (tais como velocidade de cópia memória-memória ou entrada-saída do kernel). Micro-benchmarks tornam mais fácil identificar pontos fortes e fracos de sistemas, mas eles usualmente não provêm uma boa indicação abrangendo todo o desempenho do sistema.

Eles podem ser considerados como um subconjunto de benchmarks sintéticos, entretanto, eles não tentam modelar qualquer workload real.

Micro-benchmarks têm dois problemas maiores. Primeiro, é fácil distorcer resultados usando micro-benchmarks, pois como não há um conjunto padrão de micro-benchmarks, muitos pesquisadores escrevem seu próprio conjunto para mostrar como eles melhoraram algum aspecto do desempenho do sistema. O que não é mostrado é se este melhoramento degrada outro aspecto do desempenho do sistema. O outro principal problema com micro-benchmarks é que eles não realizam trabalho real e não fazem modelagem de um workload real. Uma combinação de operações irá resultar em comportamento diferente do que a mesma operação repetida novamente.

Micro-benchmarks são, entretanto, excelentes para indicar áreas potenciais para melhoramento dentro do sistema, pois eles medem aspectos específicos do sistema.

5.2 Benchmarks usados

Lmbench[McVoy96] é um conjunto ("suite") de micro-benchmarks concebido para detectar e avaliar gargalos no desempenho de sistemas operacionais, através de medidas de "throughput" e latência ao transferir dados entre o processador, os caches, a memória, a rede e o disco. O desempenho de muitas aplicações práticas é limitado por um ou mais desses parâmetros. *Lmbench* não faz medidas de MIPS, MFLOPS, do subsistema gráfico ou capacidade de multiprocessamento. Tem como objetivos ser pequeno, portável e fácil de analisar seus resultados. Cada micro-benchmark tem tipicamente menos de 6 páginas de código em C. É distribuído livremente sob o GPL da Free Software Foundation. [McVoy96] foi eleito o melhor trabalho da Usenix 96.

Exceto em dois testes, *Lmbench* procura isolar os efeitos de periféricos como discos, placas de rede, latência e colisões em redes Ethernet sobre operações de E/S, que em geral apresentam variâncias maiores que as do subsistema de memória. Uma exceção é o micro-benchmark *lmdd* que mede o throughput de acesso a disco, em operações de leitura e gravação de arquivos. A outra exceção é *bw_tcp* que opcionalmente permite a execução remota dos testes com TCP.

Hbench:OS[Seltzer97], é baseado no código do *Lmbench*, introduzindo algumas melhorias cosméticas como: execução automática dos micro-benchmarks dirigida por um "script Korn shell", melhor apresentação e tratamento estatístico dos resultados, seu armazenamento numa base de dados histórica, e alguns ajustes interessantes nas medições de tempo:

o tempo de sobrecarga para controle de cada laço é calculado e descontado das medidas; o número de iterações é calculado dinamicamente de forma que cada micro-benchmark execute por pelo menos 1 segundo, variando de dezenas a centenas de milhares de vezes; alternativamente ao método convencional de medir tempo usando a função de sistema *gettimeofday()*, é possível utilizar os contadores de hardware do Pentium com resolução de 1 clock: a instrução especial RDTSC retorna um valor de 64 bits nos registradores EAX, EDX, contendo o valor corrente do contador de clocks. A fim de preencher os caches da CPU e memória, cada teste é executado 1 vez antes de entrar no laço principal. Os testes são repetidos *niter* vezes (um parâmetro global), calculando a média e o desvio padrão das *niter* medidas (descontadas as n% piores e melhores).

Por ser mais recente e pelas características citadas acima, escolhemos os fontes do *Hbench:OS* em vez do *Lmbench* para executar os testes, exceto pelo teste *lmdd* do *Lmbench*. Posteriormente, através da documentação da nova versão do *Lmbench*, descobrimos que várias das críticas contundentes ao *Lmbench* apresentadas em [Seltzer97] eram infundadas, e achamos prudente alertar sobre o assunto ao seu eventual leitor, uma vez que *Hbench:OS* foi construído sobre *Lmbench*.

MAB [Howard88] é um benchmark sintético que consiste de um script de comandos que opera em uma coleção de arquivos constituindo um programa de aplicação. As operações têm o objetivo de representar ações típicas de um usuário médio.

MAB é uma versão modificada do benchmark Andrew desenvolvido por M. Satyanarayanan para medir o desempenho do sistema de arquivos Andrew. O benchmark opera copiando uma hierarquia de diretórios contendo o código fonte para um programa, examinando cada arquivo na nova hierarquia, lendo os conteúdos de cada arquivo copiado, e finalmente compilando o código na hierarquia copiada.

O benchmark original de Satyanarayanan usava qualquer compilador C que estava presente na máquina central ("host"), o qual tornava impossível comparar tempos de execução entre máquinas com diferentes arquiteturas ou diferentes compiladores. Para tornar os resultados comparáveis entre máquinas diferentes, [Ousterhout90] modificou o benchmark de maneira que ele sempre usava o mesmo compilador, GNU C, gerando código para uma máquina experimental chamada SPUR. [Baker96] modificou o benchmark para gerar código para a arquitetura x86 sob Linux uma vez que a arquitetura SPUR não é suportada como uma compilação alvo.

5.3 Descrição dos benchmarks

5.3.1 Micro-benchmarks

Os micro-benchmarks pertencem a 2 categorias, os que medem em Mbytes/s alguma taxa ou "bandwidth" de interesse (bw_nome_teste) e os que medem latência em microssegundos (lat_ nome_teste). No que se segue, *bufsize* é um parâmetro variável dos micro-benchmarks, tipicamente variando em potências de 2 a partir de 2 Kbytes.

bw_mem_rd — mede a taxa de leitura da memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor (mais eficiente do que endereçamento via apontador). Inclui o custo de uma adição por palavra lida.

bw_mem_wr — mede a taxa de escrita na memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor. Inclui o custo de uma adição por palavra escrita.

bw_bzero — mede a taxa de escrita na memória usando a função de sistema *bzero*() ou *memset*().

bw_mem_cp — taxa obtida ao copiar dados entre dois buffers de memória, alinhados (libc aligned) ou não em fronteira de página (libc unaligned), através da libc, em unidades de *bufsize* Kbytes.

bw_file_rd — mede a taxa de releitura de um arquivo de 8MB já cacheado em memória (no "buffer cache"), em unidades do parâmetro *bufsize* Kbytes.

bw_pipe — mede a taxa obtida ao transferir dados através de um pipe entre dois processos em unidades de *bufsize* Kbytes.

bw_tcp — mede a taxa obtida ao transferir dados em unidades de *bufsize* Kbytes entre dois processos conectados via TCP, utilizando loop local.

lmdd — mede taxas de transferência para copiar seqüencialmente um arquivo especificado de entrada para um arquivo especificado de saída, em unidades de *bufsize* Kbytes; quando o parâmetro de entrada é *internal* o arquivo de entrada é /dev/null, o que equivale à criação e escrita do arquivo de saída; quando o parâmetro de saída é *internal* o arquivo de saída é /dev/null, o que equivale à leitura do arquivo de entrada; se a opção *sync* (ou *fsync*) for escolhida a medição de tempo termina após conclusão da chamada de sistema *sync* (ou *fsync*). Se a opção *rand* for escolhida as operações são

feitas aleatoriamente sobre o(s) arquivo(s). Todas as medidas foram feitas com a opção *sync*.

lat_pipe mede a latência para passar ida e volta um token de 1 byte através de um pipe entre 2 processos.

lat_connect mede a latência para estabelecer uma conexão TCP local.

lat_tcp mede a latência para enviar ida e volta 1 byte entre 2 processos locais via TCP.

lat_udp mede a latência para enviar ida e volta 1 token contendo um inteiro entre 2 processos locais conectados via UDP.

lat_ctx2 mede a latência média do chaveamento entre 2 processos ao passar um token através de uma seqüência de pipes conectando n processos (n varia de 2 a 20). A latência do chaveamento é definida como o tempo para mudar o contexto de hardware, atualizar o hardware TLB da CPU e possíveis conflitos de caches. É descontado o tempo de latência de passar o token via pipe. Este é um dos testes mais complexos e como os conflitos de cache são irreproduzíveis, é o que apresenta maior variância.

lat_proc mede a latência para criar um processo de 3 formas:

- null — criação de um processo nulo via *fork*();
- simple — criação de um processo via *fork*() seguida de *execve*() do programa *hello_world*;
- sh — criação de um processo via *fork*() seguida de *execlp*() para execução de *hello_world* através de /bin/sh.

lat_syscall latência para executar as seguintes chamadas de sistema: *getpid*, *gettimeofday*, *sbrk* e *write* em /dev/null.

lat_fs mede a latência de operações de metadados sobre arquivos: criação (*create*), remoção na ordem de criação (*delforw*), remoção na ordem reversa (*delrev*) e remoção em ordem aleatória (*delrand*); parâmetro: tamanho do arquivo em Kbytes (0 a 10KB).

lat_fslayer latência do sistema de arquivos, i.e., da camada vnode/VFS; medida pelo tempo para escrever um byte em /dev/null.

5.3.2 MAB

A entrada para o MAB é uma subárvore de arquivos fontes consistindo de aproximadamente 70 arquivos (Figura 5.1). Os arquivos formam o código fonte de um programa de aplicação e têm um tamanho total de aproximadamente 200 KB. O benchmark consiste de cinco fases distintas:

I MakeDir Constrói uma subárvore alvo que é idêntica à subárvore fonte.

Criação de diretórios a partir de tmp:

mkdir testfs1/include testfs1/include/sys testfs1/include/netinet

II Copy Copia cada arquivo da subárvore fonte para a subárvore destino.

Grande volume de movimentação de dados: copiar todos os fontes da subárvore *fscript* para a subárvore *testfs1*:

cp fscript/DrawString.c testfs1/DrawString.c

.

.

.

cp fscript/wait.h testfs1/include/sys/wait.h

III ScanDir Atravessa a subárvore alvo e examina o estado de cada arquivo.

Obtém estatísticas de cada diretório que compõe a subárvore testfs1, recursivamente:

find . -print -exec ls -l /*imprime as informações relativas a cada arquivo da subárvore */

du -s *; /* dá o número de blocos ocupado por cada arquivo da subárvore */

IV ReadAll Examina cada byte de cada arquivo na subárvore alvo.

Examina cada arquivo na subárvore primeiro procurando pela palavra inexistente "kangaroo" e em seguida contando quantas palavras cada arquivo

possui. Essas tarefas são proporcionais ao tamanho do arquivo e são recursivas.

find . -exec grep kangaroo

find . -exec wc

V Make Compila e liga todos os arquivos na subárvore alvo.

Computacionalmente intensiva: compila e liga os fontes do programa de aplicação da subárvore testfs1 através de um makefile presente no diretório testfs1.

cd testfs1; **make**;

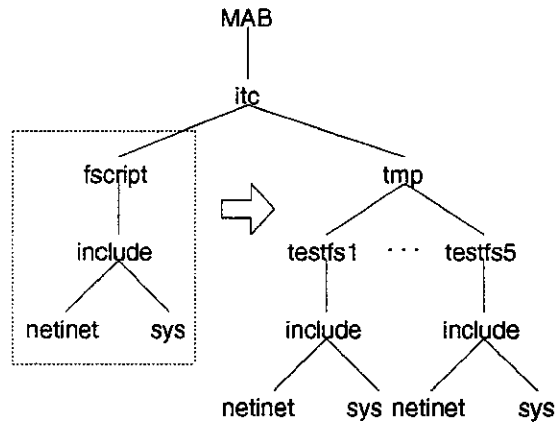


Figura 5.1: Árvore de diretórios do MAB

A saída do MAB consiste do tempo gasto em cada fase. A Fase I mede a taxa de transferência ("throughput") de escrita de meta-dados; a Fase II mede o throughput de escrita e leitura de dados e meta-dados. A Fase III consiste de leituras de meta-dados, as quais são realizadas no buffer cache, uma vez que após a cópia (Fase II) a hierarquia completa está no buffer cache, enquanto a Fase IV consiste basicamente de leituras de dados, também realizadas no buffer cache. A Fase V consiste de leituras e escritas de meta-dados e dados; as leituras são realizadas na cache, enquanto as escritas são assíncronas. Dividindo-se o benchmark em fases e apresentando o tempo gasto em cada fase, pode-se obter uma idéia de como partes diferentes do sistema de arquivos executa, tais como faixa

de passagem do buffer cache, tamanho mínimo do buffer cache, e throughput para escritas de meta-dados.

5.4 Ambiente de execução dos benchmarks

A fim de executar o shell script do Hbench:OS e algumas chamadas de sistema existentes no UNIX mas não no Plan 9, e com o objetivo de não alterar o código fonte dos benchmarks, de forma a não introduzir distorções nas medidas, decidimos por utilizar o módulo APE - Ansi/Posix Environment do Plan 9 [Trickey95], que através do comando *pcc* emula o ambiente UNIX para programas fontes escritos de acordo com o padrão ANSI/POSIX. Exceto pelo requerimento adicional, de que todo programa C sob Plan 9 exige que procedimentos externos invocados pelo programa sejam definidos via *templates* (o que implica que todo o software fonte do Plan 9 usa tipos fortes, [Pike95]), nenhuma modificação foi necessária nos fontes do Hbench:OS. Em [Trickey95] é explicitado que: "*usando APE causa compilações mais lentas e execuções marginalmente mais lentas*". Mesmo assim, nos preocupamos com esse uso, e codificamos diretamente sob Plan 9 o micro-benchmak lat_proc, tendo obtido resultados melhores que 10% com Plan 9 nativo sobre APE, apenas no teste *null* (criação de um processo nulo via fork(): 850 μs, Plan 9 x 1482 μs, APE). Dois testes não críticos, que usavam a chamada de sistema *mmap()*, inexistente no Plan 9, foram descartados do benchmark. No teste *lat_proc*, com opção *sbrk*, utilizamos a chamada de sistema *sbrk(1)* em vez da original *sbrk(0)*, pois Linux otimiza esta última, retornando um tempo inferior a 0.17 μs, já que o espaço de dados do processo não é alterado: apesar da chamada *sbrk(1)* não ser idempotente, julgamos que a comparação com Plan 9 seria mais justa.

Para medir tempo utilizamos a opção *counter* que lê o contador de ciclos de relógio da CPU via RDTSC, não tendo sido observadas discrepâncias em relação ao método convencional com *gettimeofday()*, uma vez corrigida a situação de erro relatada a seguir.

Para implementar esta opção no Plan 9, foi preciso escrever uma pequena rotina em Assembler (5 instruções), chamada por uma função "stub" em C, que retorna o valor de 64 bits do contador numa variável C de 64 bits. Funcionou perfeitamente em testes isolados com *sleep()* para aferir a correção do valor retornado. No entanto, nos testes de bandwidth do script, cerca de 10% das medidas retornavam o valor final do contador $\leq$ ao valor inicial, o que é claramente impossível. Para piorar a situação, a rotina *printf* do Plan 9, usada na depuração, mostra incorretamente valores de 64 bits. Após muita especulação, descobrimos que o erro estava no compilador C do Plan 9, na definição das duas variáveis de 64 bits para armazenar os valores inicial e final do contador, e do tipo retornado pela função stub mencionada e que estavam sendo definidos através de dois *typedefs* encadeados no *Hbench:OS*. O problema simplesmente desapareceu, ao redefini-las da seguinte forma:

```
unsigned long long start_clk, stop_clk;    /*extensão do C para variáveis de 64 bits*/
unsigned long long cycle(void);            /*função para obter o valor do contador*/
{
   return _RDTSC();                        /*chamada da rotina em Assembler*/
}
```

5.4.1 MAB

Tivemos que fazer algumas alterações no script do MAB, pois o Plan 9 não possui o comando *find* do UNIX. Para que o benchmark não ficasse diferente nos dois sistemas, de forma a comprometer a análise do tempo de execução, fizemos as mesmas alterações no Linux e SunOS.

Com relação à fase V fizemos alguns comentários nos programas fontes em funções que não existem no Plan 9. Executamos o benchmark no Linux com e sem as alterações feitas no Plan 9 e obtivemos o mesmo tempo, o que nos levou a concluir que as alterações realizadas não influem no tempo de execução do benchmark.

As alterações foram realizadas na fases III, IV e V, sendo que nas fases III e IV foram substituídos os comandos abaixo listados por outros que têm a mesma função. E na fase V as funções listadas foram comentadas.

III ScandDir **ls** -l ‘**du** -a | **awk** ‘{print $$2} ‘
du -s *;

IV ReadAll **sed** -n “/kangaroo‘ ’**du** -a | **awk** “{print $$2}“‘
wc “**du** -a | **awk** “{print $$2}“‘

V Make *getpagesize, mmap, killpg, fwrite, setsockopt, index, vfork.*

5.5 Comentários sobre os resultados dos testes

Os testes foram feitos em um PC Pentium de 200MHz, com 64 MB de memória RAM, disco Quantum Fireball de 3 GB com 128 KB de cache, placa de rede Ethernet PCI 10MB/s e em um PC Pentium MMX de 233 Mhz, com 64MB de memória RAM, disco Fujitsu Ultra DMA de 3GB (com 0 KB de cache), placa de rede Ethernet PCI 10MB/s, ambos com Plan 9 e Linux Redhat 5.1 (kernel 2.0.34) instalados em partições distintas do disco e em uma SparcStation 1+ Modelo 4/65 de 25MHz, com 28 MB de RAM e sistema operacional SunOS versão 4.1.3.

Foram feitas 50 medidas para cada micro-benchmark e a razão (desvio padrão)/média obtida nos testes, foi tipicamente inferior a 1 % (exceto no teste lat_ctx2) e por isso não está sendo relatada.

Como as arquiteturas SPARC e PC são muito diferentes, além de haver um fator de 8 no valor do relógio e a normalização dos resultados por este fator, consideramos a comparação dos números da SparcStation com os do Plan 9 e Linux aproximada e, por essa razão, apenas alguns comentários gerais serão feitos sobre os mesmos.

Na normalização dos resultados da SPARC os testes de transferência foram multiplicados por 8 e os testes de latência foram divididos por 8, pois:

$$\frac{clock\ SPARC}{clock\ PC} = \frac{25MHz}{200MHz} = 1/8 = 0.125$$

5.5.1 Testes de memória

bw_mem_rd Figura 5.2

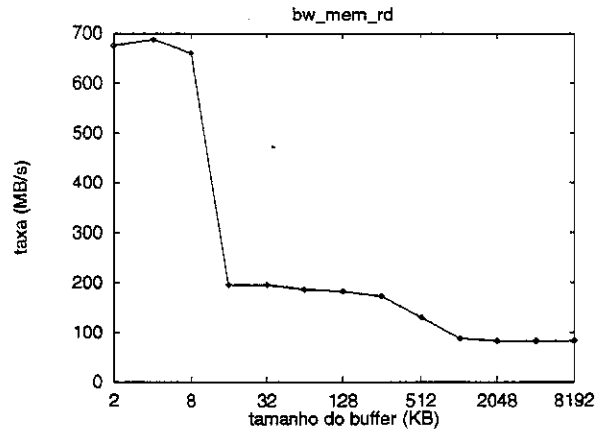
bw_mem_wr Figura 5.3

bw_bzero Figura 5.4

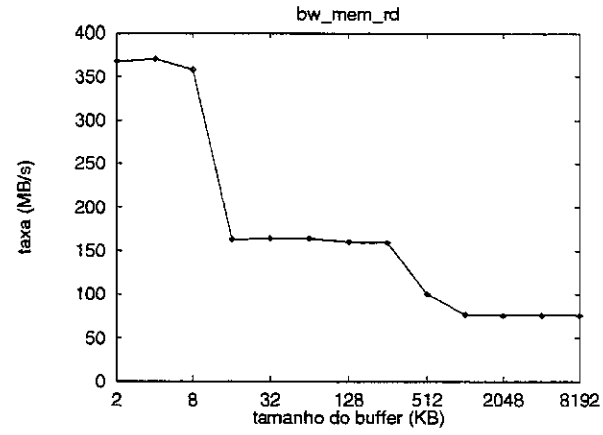
bw_mem_cp Figura 5.5

É interessante observar que enquanto os dados estão no cache L1 da CPU (8KB), Plan 9 tem um desempenho de leitura 85% maior que linux, aproximando-se para buffers $>$ 16KB. O de escrita(bw_mem_wr), no entanto, é cerca de 1/3 do Linux no cache L1 e equivalente a partir de buffers $\geq$ 32KB. No caso de bw_bzero o desempenho do Plan 9 é praticamente constante, sendo 1/9 do Linux no cache L1 e 40% melhor que Linux a partir de 512KB. Já no teste de cópia de dados (bw_mem_cp) Plan 9 tem 1/5 de desempenho do Linux com buffer $<$ 8KB mas cerca de 25% superior a partir de 128KB.

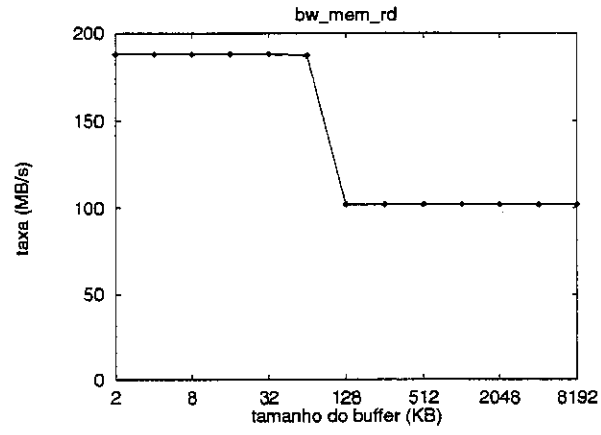
Nos testes de transferência de dados para a memória, SunOS mostrou-se inferior a Linux para tamanhos de buffers inferiores a 64K e comparável para valores maiores. A cópia de memória é comparável ao Plan 9.



(a) bw_mem_rd-plan9

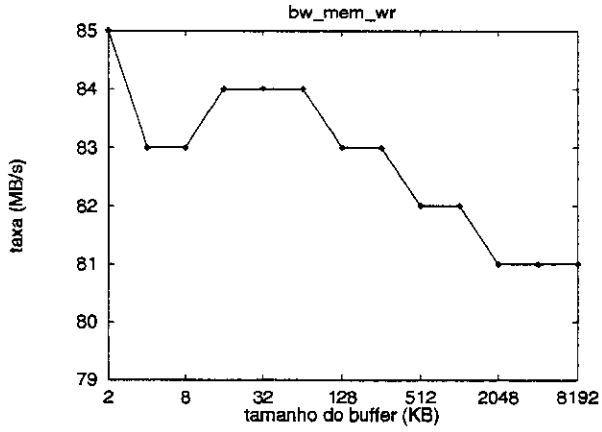


(b) bw_mem_rd-linux

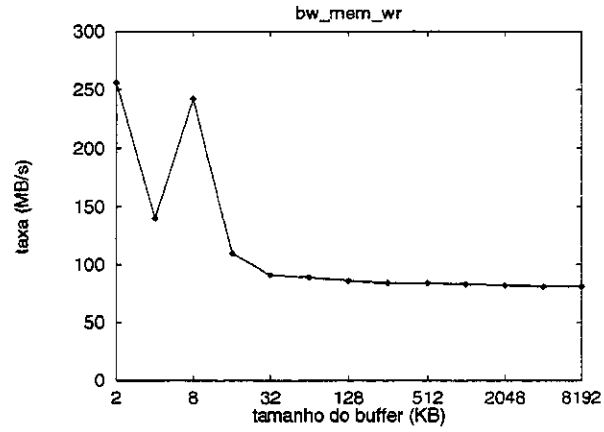


(c) bw_mem_rd-sunos

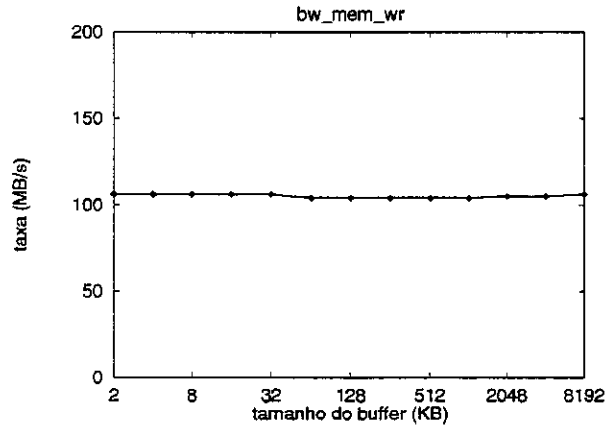
Figura 5.2: bw_mem_rd: mede a taxa de leitura da memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor.



(a) bw_mem_wr-plan9

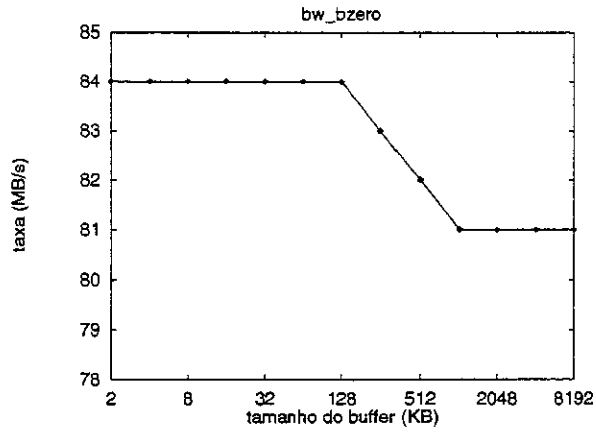


(b) bw_mem_wr-linux

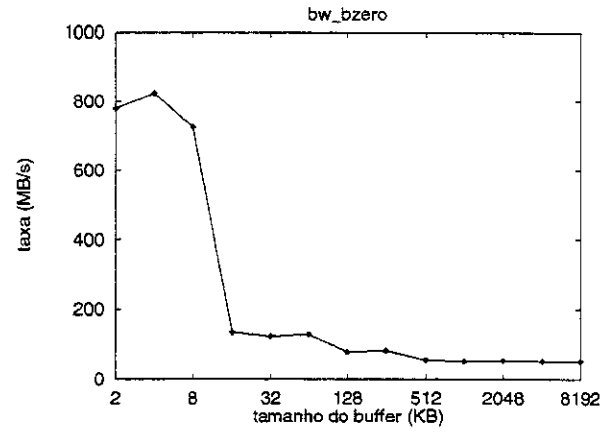


(c) bw_mem_wr-sunos

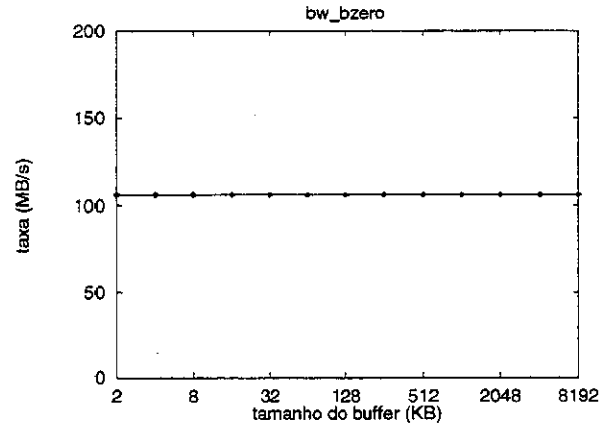
Figura 5.3: bw_mem_wr: mede a taxa de escrita na memória em unidades de *bufsize* Kbytes, através de um laço "unrolled" e endereçamento via deslocamento de vetor.



(a) bw_bzero-plan9

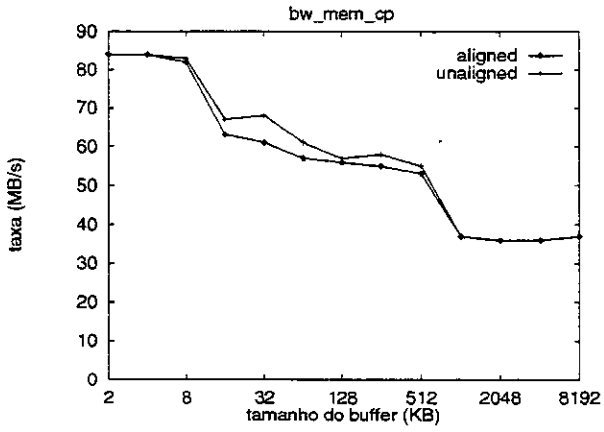


(b) bw_bzero-linux

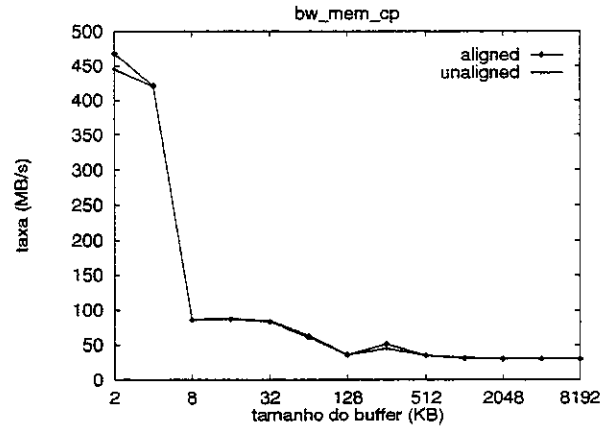


(c) bw_bzero-sunos

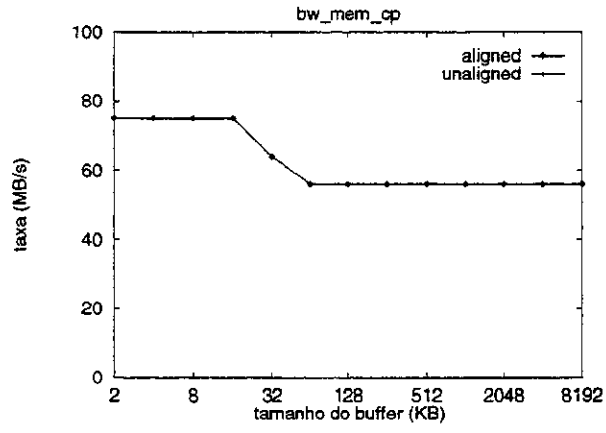
Figura 5.4: bw_bzero: mede a taxa de escrita na memória usando a função de sistema *bzero*() ou *memset*().



(a) bw_mem_cp-plan9



(b) bw_mem_cp-linux



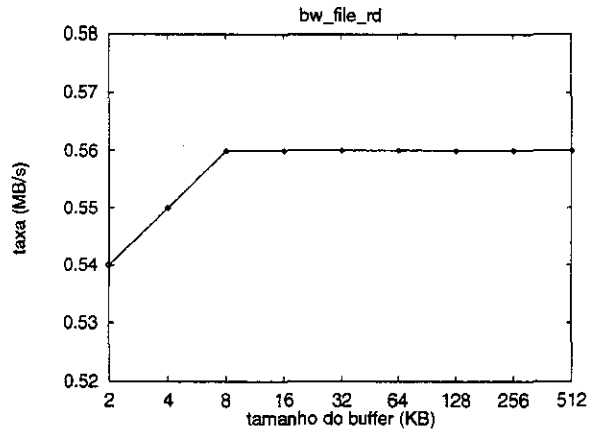
(c) bw_mem_cp-sunos

Figura 5.5: bw_mem_cp: taxa obtida ao copiar dados entre dois buffers de memória, alinhados (aligned) ou não em fronteira de página (unaligned), através da libc, em unidades de *bufsize* Kbytes.

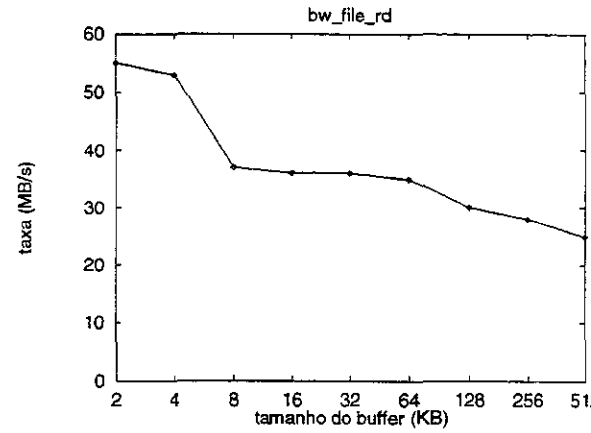
5.5.2 Teste de leitura do buffer-cache

bw_file_rd Figura 5.6

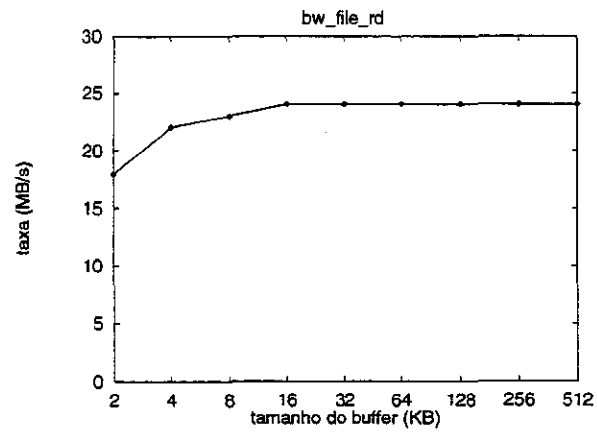
Neste teste fica clara a insuficiência de gerenciamento do "buffer cache" do Plan 9, pois o arquivo de 8MB deveria estar todo na memória durante o teste. Linux consegue taxas de 50MB/s, enquanto Plan 9 apresenta um valor constante em torno de 0.55MB/s, menor inclusive, do que o apresentado por *lmdd* abaixo, ao ler um arquivo não cacheado de 100MB. Linux é também claramente superior a SunOS.



(a) bw_file_rd-plan9



(b) bw_file_rd-linux



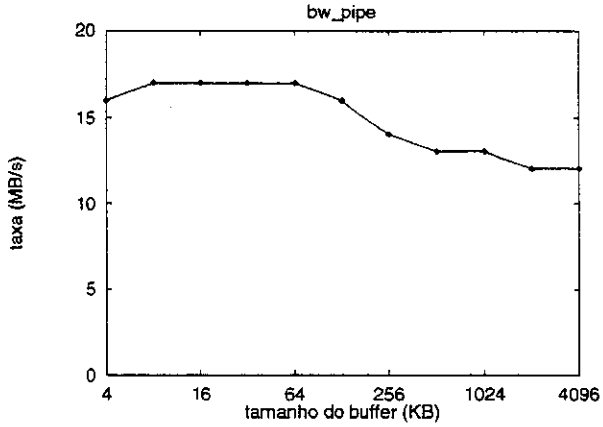
(c) bw_file_rd-sunos

Figura 5.6: bw_file_rd: mede a taxa de releitura de um arquivo de 8MB já cacheado em memória (no "buffer cache"), em unidades do parâmetro *bufsize* Kbytes.

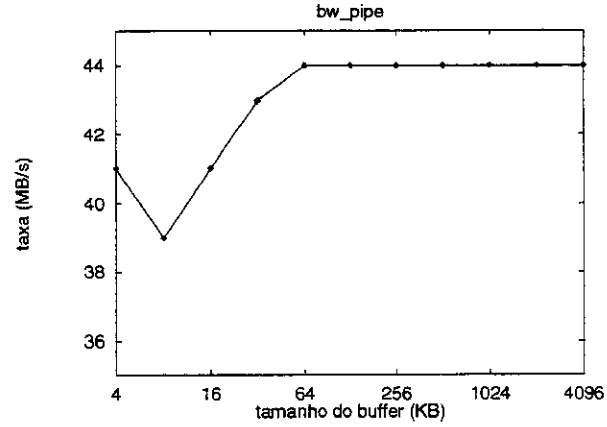
5.5.3 Transferência de dados via pipe

bw_pipe Figura 5.7

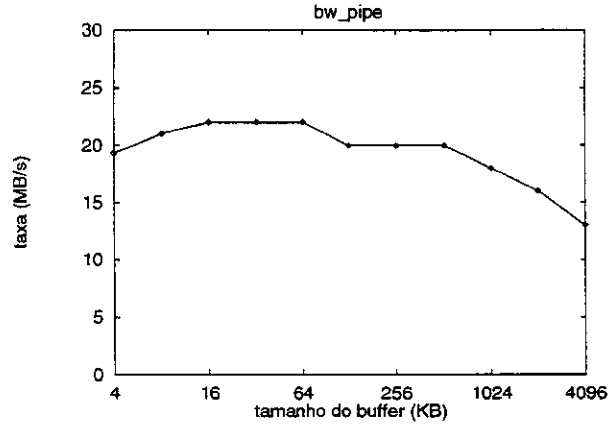
Linux é cerca de 2.5 vezes mais veloz do que Plan 9 e 2 vezes mais veloz que SunOS para todos os tamanhos de buffers.



(a) bw_pipe-plan9



(b) bw_pipe-linux



(c) bw_pipe-sunos

Figura 5.7: bw_pipe: mede a taxa obtida ao transferir dados através de um pipe entre dois processos em unidades de *bufsize* Kbytes.

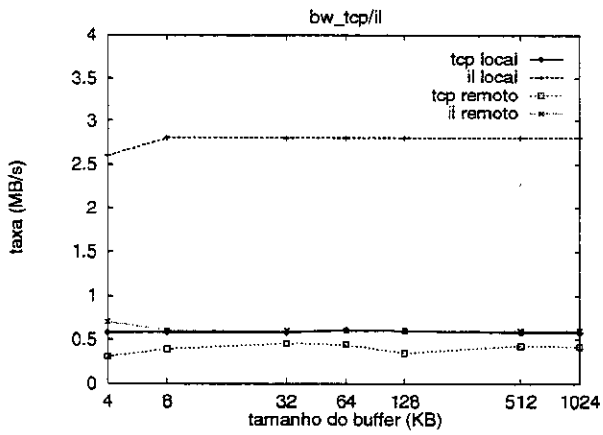
5.5.4 Transferência de dados via TCP

bw_tcp Figura 5.8

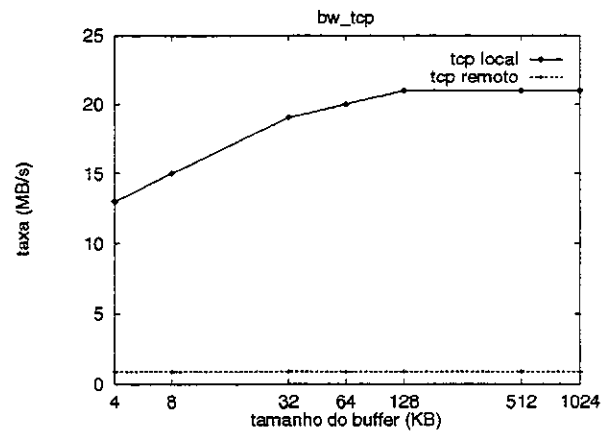
Aqui a diferença é maior ainda: Linux é pelo menos 20 vezes mais veloz do que Plan 9. É interessante notar a evolução contínua do Linux: conforme observado em [Baker96] a razão bw_pipe/bw_tcp (buffer de 48KB) era 4.8 enquanto para FreeBSD era 1.48, o que sugere uma implementação inferior de TCP no Linux do que no FreeBSD. No teste bw_tcp, FreeBSD foi 2.6 vezes mais veloz que Linux (na época, o kernel 1.2.8 tinha uma janela TCP de apenas 1 pacote). Nos nossos testes, bw_pipe/bw_tcp = 2.2, o que sugere uma melhoria de pelo menos 100% no desempenho do TCP, muito provavelmente obtida, pois se normalizarmos pelo valor do clock, bw_tcp [K.LAY96] x 2= 6.3MB/s enquanto nos nossos testes bw_tcp=20MB/s.

As conexões TCP e IL no Plan 9 utilizam a mesma sintaxe, de forma que também fizemos o teste de bandwidth para IL. IL é um protocolo voltado para transações RPC, do tipo pedido/resposta, e não para transmissão de dados em massa como TCP; é no entanto confiável e garante sequenciamento de mensagens. Esse teste apresentou razões desvio padrão/média relativamente grandes (em torno de 10%). Apesar de ter uma janela de 10 pacotes, IL não tem controle de fluxo, descartando pacotes que ultrapassam a janela e pedindo retransmissão. Isso talvez explique as variâncias obtidas. IL mostrou-se 5 vezes mais veloz que TCP no Plan 9, mas mesmo assim 6 vezes mais lento que TCP no Linux, para comunicação local. Na comunicação remota, IL mostrou-se 2 vezes mais veloz que TCP no Plan 9 e 30% mais lento que TCP no Linux. (IL restringe o tamanho máximo de um pacote em 32KB). Um teste comparativo melhor seria o de latência, entre IL e TCP ou IL e UDP, comentados abaixo.

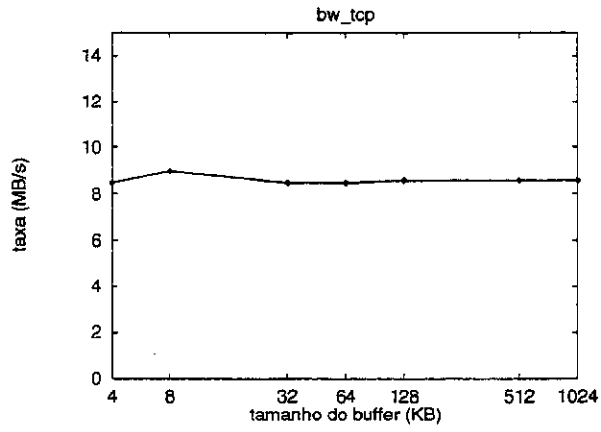
Com relação a SunOS, Linux chega a ser 2 vezes mais veloz.



(a) bw_tcp_il-plan9



(b) bw_tcp-linux



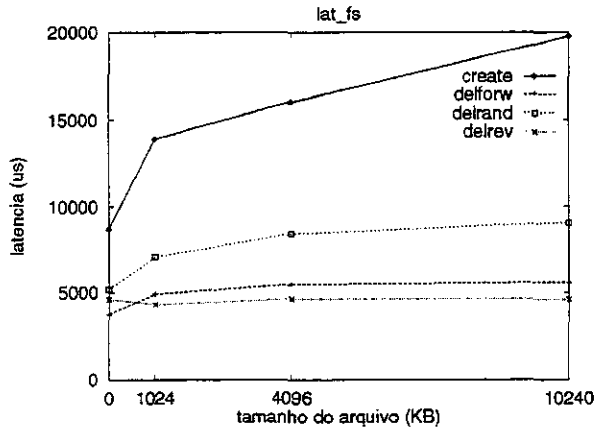
(c) bw_tcp-sunos

Figura 5.8: bw_tcp_il: mede a taxa obtida ao transferir dados em unidades de *bufsize* Kbytes entre dois processos conectados via TCP/IL.

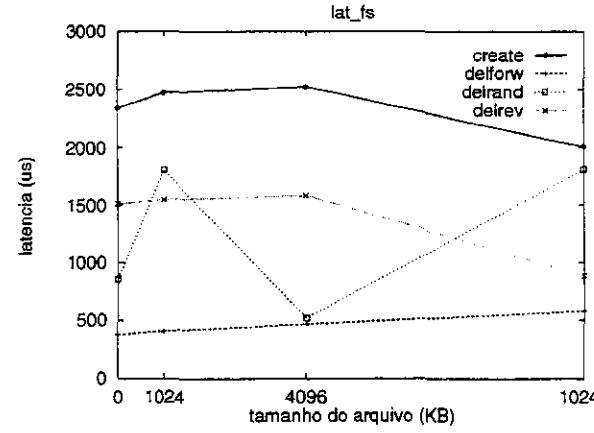
5.5.5 Operações sobre metadados de arquivos

lat_fs Figura 5.9

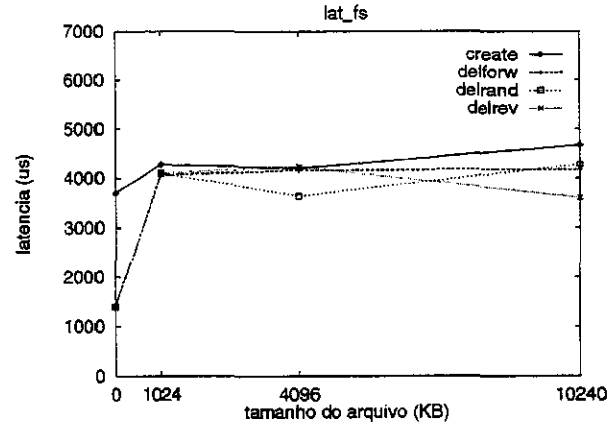
Neste teste os resultados não são comparáveis, pois essas operações são, por padrão, assíncronas no Linux (isto é, só vão para o disco quando expira o temporizador de escrita de blocos "sujos" do buffer cache), e síncronas (por razões de segurança) na maioria dos sistemas, inclusive Plan 9, de forma que Linux leva uma vantagem não justificável nesse teste. Plan 9 tem tempos normalizados comparáveis aos de Solaris e FreeBSD em [Baker96].



(a) lat_fs-plan9



(b) lat_fs-linux



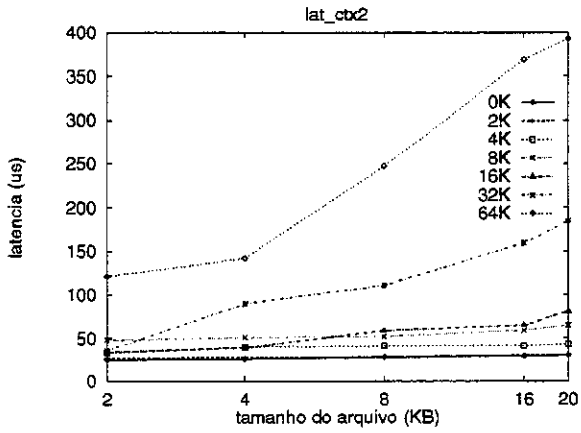
(c) lat_fs-sunos

Figura 5.9: lat_fs: mede a latência de operações de metadados sobre arquivos em Kbytes(0 a 10KB).

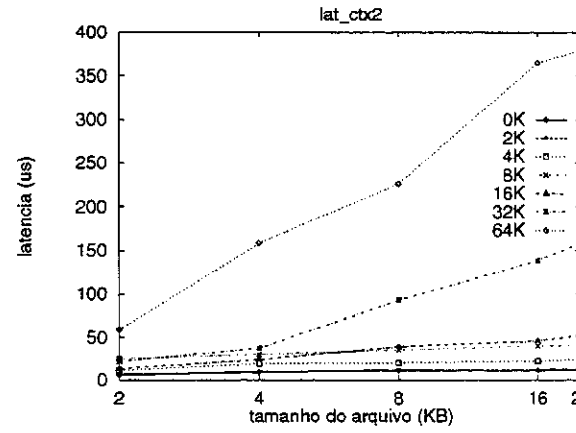
5.5.6 Latência de mudança de contexto

lat_ctx2 Figura 5.10

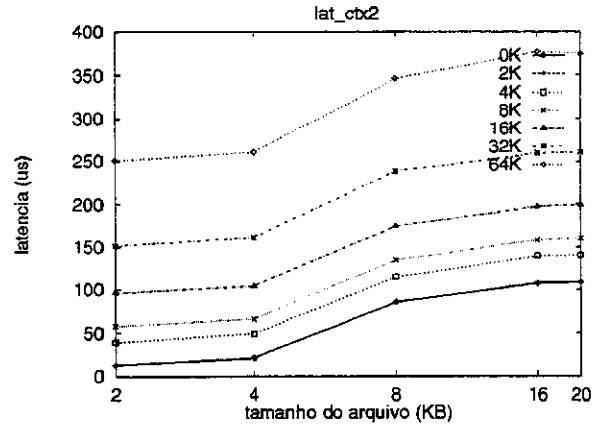
Conforme previsto, este é o teste que apresentou maiores variâncias; mesmo assim, no Linux, a razão desvio padrão/média foi inferior a 6% para processos menores que 8KB, chegando a 25% para processos até 64KB, não havendo correlação entre esse valor e o número de processos. Plan 9 apresentou para esses valores 3% e 12%, respectivamente. Com esta ressalva em mente, os números do Linux são melhores, especialmente com poucos processos (2 a 4) embora a diferença deixe de ser significativa quando há mais de 8 processos com tamanho $\geq$32KB. Como esta é uma situação típica, não vemos, na prática, diferença de desempenho entre os dois sistemas nesse teste.



(a) lat_ctx2-plan9



(b) lat_ctx2-linux



(c) lat_ctx2-sunos

Figura 5.10: lat_ctx2: mede a latência média do chaveamento entre 2 processos ao passar um token através de uma seqüência de pipes conectando n processos (n varia de 2 a 20).

5.5.7 Outros testes de latência

Tabela 5.1

Nesses testes Linux apresenta valores em geral várias vezes menores que Plan 9, exceto no teste de criação do processo nulo, onde a diferença é cerca de 11%, sendo Plan 9 mais rápido no teste nativo mencionado, mas não quando fork é seguido de exec, provavelmente por causa do acesso ao sistema de arquivos. Em [Pike95] é argumentado que, pela rapidez de criação e chaveamento de processos, "kernel threads" são desnecessários no Plan 9. Cremos que o mesmo argumento aplica-se para Linux. Nos testes de latência com pipes Linux leva 5 vezes menos tempo e 2 vezes menos com TCP; no teste de latência com IL no Plan 9, este foi 50% mais lento que TCP no Linux. Nos testes *lat_syscall* as diferenças são maiores ainda; *getpid* é provavelmente buferizado no Linux [Baker96].

Nos testes de latência de TCP, IL e UDP o melhor tempo local do Plan 9 (lat_udp = 300μs) é cerca de o dobro do melhor tempo local do Linux (lat_udp = 159μs). Já, o melhor tempo remoto do Plan 9 (lat_il = 443 μs) é apenas 20% maior que o melhor tempo remoto do Linux (lat_udp = 374μs).

Os valores normalizados mostram vantagem do SunOS sobre Linux em alguns testes: lat_proc, lat_connect, lat_udp. Pelas razões citadas anteriormente, esses números devem ser vistos com suspeita.

Teste	Parâmetro	Plan 9	Linux	SunOS
lat_proc	null	1480	1270	2980
lat_proc	simple	10870	6240	4200
lat_proc	shell	75840	21990	11320
lat_pipe	-	117	23	88
lat_connect (local)	-	18270	370	250
lat_connect (remoto)	-	14860	740	na
lat_tcp (local)	-	460	230	700
lat_tcp (remoto)	-	570	430	na
lat_il (local)	-	360	na	na
lat_il (remoto)	-	450	na	na
lat_udp (local)	-	300	160	150
lat_udp (remoto)	-	470	380	na
lat_syscall	getpid	125	1.1	3.8
lat_syscall	gettimeofday	60	2.1	5.2
lat_syscall	sbrk(1)	5.3	1.4	4.5
lat_syscall	write(/dev/null)	8.0	1.8	13.8
lat_fslayer	-	8.6	1.9	13.8

Tabela 5.1: Medidas de latência local/remota (microssegundos)

5.5.8 E/S em disco

LMDD Local

Figura 5.11 e Tabela 5.2

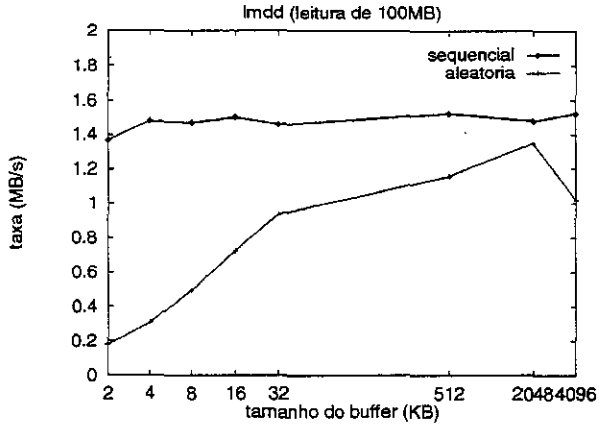
Todos os testes foram feitos com um arquivo de tamanho 100MB. Nesse caso a memória interna é insuficiente para cachear todo o arquivo, mas mesmo assim tomamos o cuidado de reiniciar a máquina a cada teste, a fim de garantir que o cache não seria reutilizado pelo teste seguinte. Linux consegue ler sequencialmente do disco um arquivo de 100MB à taxa de 8.33MB/s (bloco de 2KB) contra 1.52MB/s no Plan 9, ou seja, 5 vezes mais veloz. O disco utilizado, um Quantum Fireball ST 3.2AT de 3GB, gira a 5400 rpm, tem um tempo de seek ("posicionamento do braço") médio de 10ms, e um número variável de setores/trilha, variando de 277 a 154 (dados obtidos em *www.quantum.com*), fazendo com que a taxa nominal de transferência varie de 6.8 MB/s a 12.2 MB/s. Possui um cache interno de 128KB, (cerca de 1 trilha) e é capaz de transferir para o barramento IDE a 16.7 MB/s. Como a partição Linux começa a partir dos primeiros 25% do espaço total do disco, estimamos que naquele ponto a taxa nominal seja cerca de 9.2MB/s. Desta forma Linux consegue extrair na leitura sequencial cerca de 90% da taxa nominal do disco. Na leitura aleatória o desempenho do Linux se aproxima ao da leitura sequencial com blocos de 32 KB. Com blocos $\geq$ 32KB, o desempenho é melhor do que na leitura sequencial porque parte do arquivo está cacheada na memória e nem sempre é preciso ir ao disco. Nos testes de escrita usou-se blocos de 8KB. Na Tabela 5.2 pode-se ver que na escrita sequencial Linux atingiu 4.6MB/s, ou seja, 50% da taxa nominal do disco e 13 vezes mais veloz que Plan 9, e na escrita aleatória foi 6 vezes mais veloz. Este último teste é dominado pelo tempo de seek, fazendo no total 12800 seeks; como no Linux ele durou 116.3 segundos, o tempo médio por operação foi de 9.8 ms, próximo ao tempo médio de seek especificado pelo fabricante do disco . O fato da partição Plan 9 estar em trilhas mais internas que a partição Linux, fazendo com que as taxas nominais de transferência para o Plan 9 sejam menores em até 25% não explicam tal diferença. Um driver de disco pouco otimizado, a ausência de leitura antecipada e um buffer cache pequeno são melhores candidatos para explicar esta diferença.

Os coeficientes de variação também não foram apresentados por serem tipicamente inferiores a 1%.

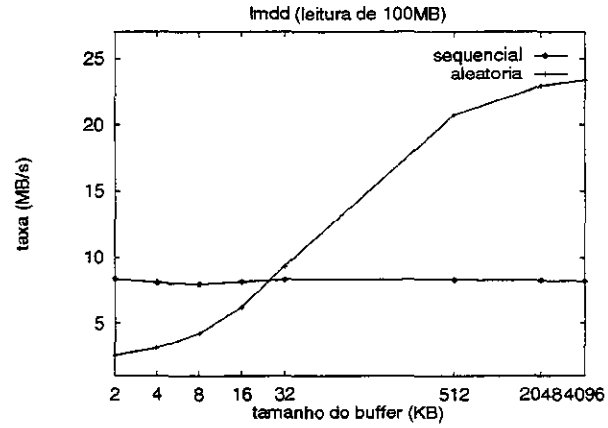
Embora não diretamente comparáveis por se tratar de hardware bastante diferente e um relógio de CPU bastante lento (25MHz), incluímos na Tabela 5.2 e Figura 5.11 os resultados dos testes para a SparcStation 1+.

Tipo	Plan 9	Linux	SunOS
seqüencial	0.36	4.64	1.75
aleatória	0.15	0.86	0.7

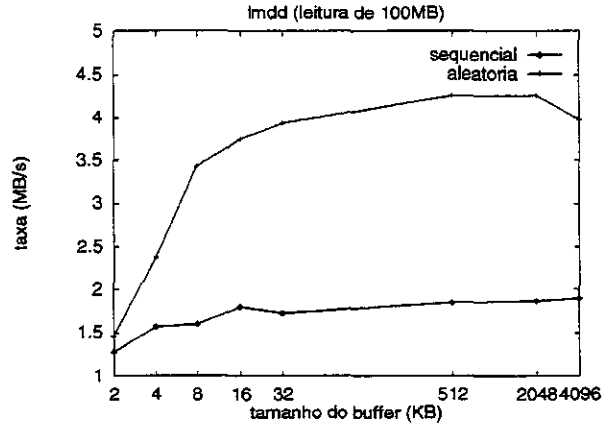
Tabela 5.2: Escrita LMDD local (MB/s)



(a) lmdd-plan9



(b) lmdd-linux



(c) lmdd-SunOS

Figura 5.11: Leitura LMDD local (MB/s):mede taxas de transferência para copiar seqüencialmente um arquivo local de tamanho 100MB, em unidades de *bufsize* Kbytes.

LMDD Remoto

Figura 5.12 e Tabela 5.3

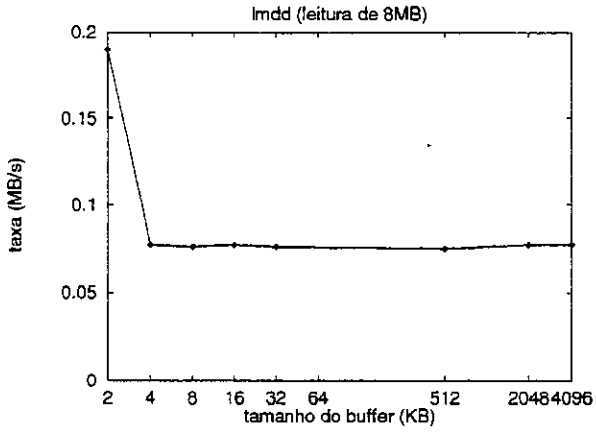
Na execução do LMDD remoto, usamos um arquivo de tamanho igual a 8MB, de modo que esse arquivo estivesse por completo na memória RAM ("warm cache"), pois a máquina com menor tamanho de memória RAM é a SparcStation, a qual tem 28MB. Para que esse arquivo fosse colocado na memória primeiro executamos a escrita LMDD a partir do cliente no servidor, pois dessa maneira o arquivo estaria cacheado no servidor. Usamos o arquivo cacheado para evitarmos o gargalo de leitura em disco, de modo que as medidas resultantes se referem somente sobre o protocolo de comunicação entre os sistemas.

Em seguida efetuamos a leitura do arquivo no servidor a partir do cliente, em vários tamanhos de bloco, sincronamente. Pelos gráficos temos que Plan 9 - Linux tem o melhor desempenho, seguido por Plan 9 - Plan 9, Linux - Linux, Linux - SunOS, Plan 9 - SunOS. Podemos concluir que o fato de Plan 9 - Linux aparecer em primeiro lugar seguido de Plan 9 - Plan 9, deve-se ao protocolo IL que é melhor do que o NFS. Plan 9 - Linux em primeiro ao invés de Plan 9 - Plan 9, deve-se ao fato do arquivo está cacheado no Linux, mas não está no Plan 9, uma vez que um terminal Plan 9 não cacheia dados. Não temos explicação porque Plan 9 - SunOS aparece com o pior desempenho, pois achavámos que Linux - Linux e Linux - SunOS surgiriam em último.

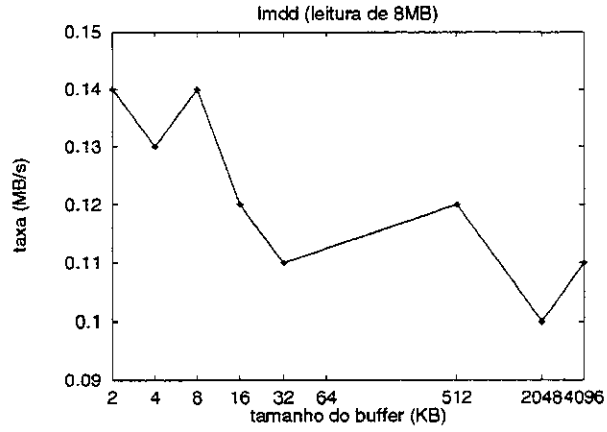
Pelos resultados dos testes com LMDD remoto, pode-se comprovar que 9P+IL são protocolos superiores ao NFS. A distribuição do Plan 9 não contém um cliente NFS, ou seja, um NFS para permitir que uma máquina Plan 9 acesse arquivos em máquina UNIX remotas usando NFS.

IL Plan 9-Plan 9	U9FS Plan 9-Linux	U9FS Plan 9-SunOS	U9FS Linux-Linux	NFS Linux-SunOS
0.81	4.17	0.27	4.14	0.02

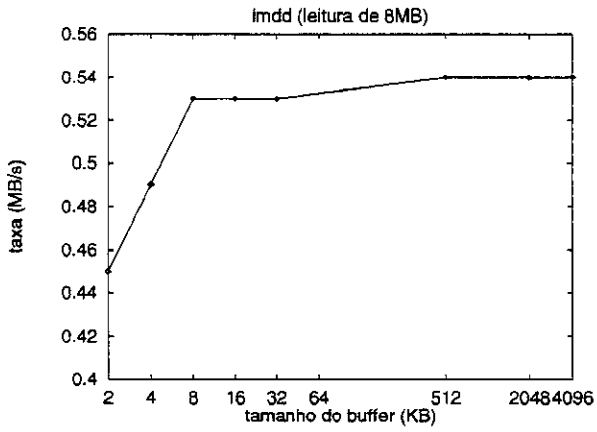
Tabela 5.3: Escrita LMDD remoto (MB/s)



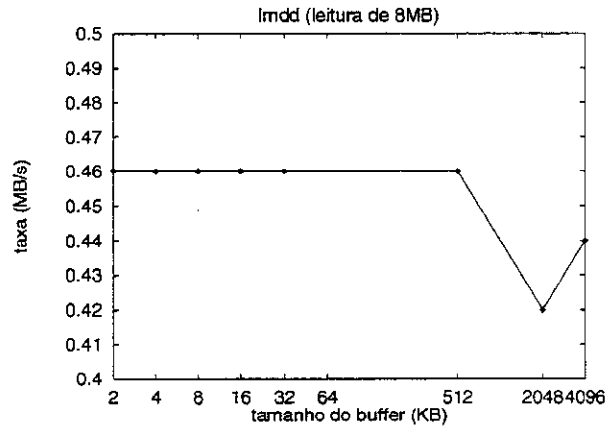
(a) lmdd-plan9_SunOS



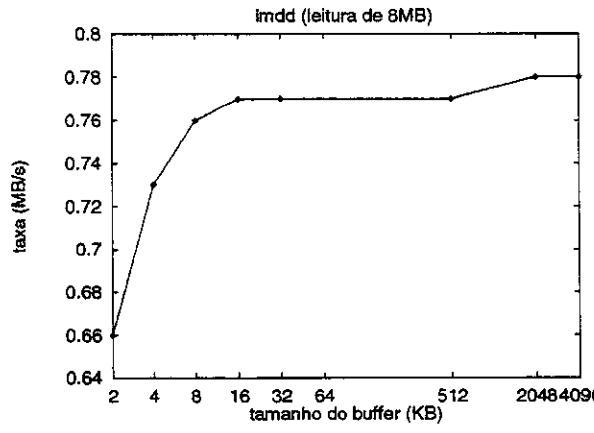
(b) lmdd-linux_SunOS



(c) lmdd-plan9_plan9



(d) lmdd-linux_linux



(e) lmdd-plan9_linux

Figura 5.12: lmdd:U9FSxNFS: mede taxas de transferência para copiar seqüencialmente um arquivo remoto de tamanho 8MB, em unidades de *bufsize* Kbytes.

5.5.9 MAB

Local

Devido ao fato do processador SPARC ser muito lento (em MHz), normalizamos as medidas no SunOS pelo valor do clock, isto é, os tempos do SunOS foram multiplicados pelo fator

$$\frac{clock\ SPARC}{clock\ PC} = \frac{25MHz}{200MHz} = 1/8 = 0.125$$

No MAB local o Linux se saiu melhor do que Plan 9 e SunOS em todas as fases.

Fase III e IV: A árvore está cacheada no Linux, enquanto não está no Plan 9. Por esta razão, Linux é muito mais rápido.

Fase V: Visto que a árvore está cacheada no Linux e não está no Plan 9, podemos afirmar que Plan 9 tem um desempenho igual ao do Linux.

É importante salientar, que executamos o benchmark em cada sistema cinco vezes, e que entre cada fase a máquina foi reiniciada, de modo que o compilador não ficasse cacheado, o que daria um desempenho ainda melhor para o Linux na Fase V. Os números entre parênteses representam o coeficiente de variação em %, isto é, (desvio padrão/média)x100%

Os tempos normalizados do SunOS, por um fator de 8, estão comparáveis aos do Linux, mas devem ser vistos com suspeita. Seria interessante executar o benchmark com SunOS instalado no mesmo PC que Linux e Plan 9.

Fases	Plan 9 (PC)	Linux (PC)	SunOS (SPARC) normalizado
I Cria dir	0.3 (0)	0.1 (8.8)	0.24 (3.1)
II ls -l	4.5 (6.6)	2.3 (0.6)	1.55 (6.4)
III Cópia	14.4 (2.1)	0.2 (0)	0.19 (4.6)
IV Busca	2.6 (3.8)	1.4 (0.5)	2.14 (2.9)
V Compila	13.6 (0.7)	13.3 (1.5)	12.2 (3.2)

Tabela 5.4: MAB local (s)

Remoto

Remotamente o Linux se sai melhor porque não somente o servidor, mas também o cliente está cacheando a árvore de arquivos. Concluímos isso, porque se executarmos o LMDD remoto, duas vezes seguidas, na segunda vez a taxa de leitura chega a 37MB/s, ou seja, 4 vezes a taxa nominal do disco! Porque o desempenho é tão alto, comparado com a primeira execução, só pode advir do fato do cliente está cacheando os dados.

Fases	IL Plan 9-Plan 9	U9FS Plan 9-Linux	U9FS Plan 9-SunOS	NFS Linux-Linux	NFS Linux-SunOS
I Cria dir	0.6 (16.6)	0.4 (25)	3.1 (32.2)	0.12 (0)	2.1 (0)
II ls -l	7.2 (18)	5.7 (8.7)	20.5 (11.2)	3.6 (1.6)	25.0 (7.2)
III Cópia	1.2 (24)	2.0 (10.0)	8.49 (0.7)	0.9 (8.8)	2.9 (5.5)
IV Busca	6.3 (4.7)	6.3 (1.5)	28.3 (7.4)	3.1 (3.2)	8.9 (1.1)
V Compila	18.3 (0.01)	21.5 (5.5)	50.1 (3.8)	16.8 (12.5)	41.44 (1.2)

Tabela 5.5: MAB remoto

Quando não temos um cliente Linux, podemos ver a eficiência do protocolo IL em Plan 9 - Plan 9 comparado com Plan 9 - Linux.

Linux - SunOS tem um baixo desempenho devido ao fato do relógio do processador da SparcStation ser baixo, comparado com os relógios dos PCs sendo utilizados.

Capítulo 6

Conclusão

Nos capítulos anteriores foi apresentado o sistema Plan 9, alguns benchmarks e várias análises de medidas de desempenho realizadas com esses benchmarks nos sistemas Plan 9, Linux e SunOS. Foram discutidos os vários resultados obtidos.

Neste capítulo são apresentadas as principais dificuldades encontradas durante o desenvolvimento deste trabalho, as contribuições apresentadas por esta dissertação e são citadas algumas sugestões para trabalhos futuros. Nos comentários finais são avaliados os resultados obtidos, de acordo com os objetivos e a motivação inicialmente estabelecidos.

6.1 Dificuldades encontradas

No decorrer do desenvolvimento deste trabalho foram encontradas diversas dificuldades de ordem prática que merecem ser destacadas.

Como colocado no capítulo 4, uma das grandes dificuldades que tivemos com o Plan 9 foi a sua alta sensibilidade ao hardware sendo usado, o que nos levou a despender muito tempo na configuração das máquinas que usamos durante o trabalho.

Os benchmarks que usamos foram implementados para serem usados em sistemas UNIX e como Plan 9 não implementa todas as funções do UNIX, como por exemplo, as funções de mapeamento de páginas da memória (*mmap()*) alguns micro-benchmarks que fazem parte do suite do Hbench:OS não foram realizados.

6.2 Contribuições deste trabalho

As principais contribuições deste trabalho estão relacionadas a seguir:

- Análise, instalação personalizada e testes do sistema operacional distribuído Plan 9.

- Levantamento e análise de ferramentas modernas para avaliação de desempenho de sistemas operacionais.
- Análise comparativa de desempenho dos protocolos de comunicação dos sistemas Plan 9 e Linux, mostrando as eficiências e deficiências de cada sistema.
- Comparação de desempenho do protocolo de transporte 9P+IL com NFS.

6.3 Trabalhos futuros

Como conhecemos, através das medidas, onde Plan 9 é deficiente, poderíamos implementar ou melhorar alguma função do Plan 9, de forma que o sistema torne-se mais eficiente, como por exemplo a introdução de um buffer cache nas máquinas terminais. Essa sugestão nos foi dada por Russ Cox da Bell-Labs.

Comparação entre Plan 9 e Solaris, ambos executando sob a arquitetura PC.

O servidor de arquivos, de acordo com a documentação, cacheia extensivamente arquivos em memória RAM e possivelmente alguns dos resultados de desempenho do Plan 9 seriam mais favoráveis, caso executados num servidor de arquivos. Sugerimos, então, a instalação de um servidor de arquivos para a realização de todas as medidas de desempenho efetuadas nesta dissertação.

6.4 Comentários finais

As medidas de desempenho mostraram que os protocolos 9P/IL do Plan 9 são superiores ao NFS (além de consideravelmente mais simples).

O protocolo IL, mesmo nos testes de comunicação do tipo requisição/resposta mostrou-se inferior às implementações de TCP e UDP no Linux.

Em alguns testes os resultados foram desfavoráveis ao Plan 9 pelo fato do cliente não implementar o uso de cacheamento de arquivos, o qual só é implementado no servidor de arquivos, e provavelmente os testes de comunicação remota seriam mais favoráveis nesse caso.

Apêndice A

Terminologia e conceitos básicos

A.1 Introdução

Este apêndice apresenta alguns conceitos básicos e termos usados neste trabalho. O propósito desse apêndice é facilitar a leitura do corpo do trabalho para os leitores não familiarizados com terminologia de sistemas operacionais.

A.2 Conceitos básicos

Kernel — o kernel é o código do sistema operacional residente na memória que executa em um estado privilegiado. Um sistema distribuído como Plan 9 é composto dos kernels em cada terminal.

Aplicação — uma aplicação é um programa que um usuário executa para executar alguma tarefa. Exemplos incluem editores de texto, compiladores e simuladores.

Processo — um processo é um programa em execução. Ele pode ser um processo à nível de usuário que é parte de uma aplicação. Uma aplicação pode ser composta de vários processos.

Chamada de sistema — uma chamada de sistema é uma chamada de procedimento especial que um processo a nível de usuário faz para invocar funções do sistema operacional. Durante uma chamada de sistema o processo continua a execução dentro do kernel em um estado privilegiado. Há um conjunto bem definido de chamadas de sistema usados para abrir arquivos, ler o relógio, criar processos, etc.

Objeto objeto é um termo geral para entidades acessadas através da interface do sistema de arquivos. Ele pode referir-se a um arquivo, diretório, dispositivo ou pseudo-dispositivo.

Stateful um servidor stateful ("com estado") é um servidor que retém informação sobre operações do cliente entre requisições e usa esta informação de estado para servir requisições subseqüentes corretamente. Requisições como *open* e *seek* ("deslocamento") são inerentemente stateful, uma vez que alguém deve relembrar quais arquivos um cliente abriu, assim como o deslocamento do *seek* em cada arquivo aberto.

Stateless Em um sistema stateless ("livre de estado"), cada requisição é auto-contida, e o servidor não mantém estado persistente sobre os clientes. Por exemplo, ao invés de manter o deslocamento do *seek*, o servidor pode requerer que o cliente especifique o deslocamento para cada leitura ou escrita. Servidores stateful são mais rápidos, uma vez que o servidor pode levar vantagem de seu conhecimento de estado do cliente para eliminar grande parte do tráfego da rede. Entretanto, eles têm consistência complexa e mecanismos de recuperação de falhas. Servidores stateless são mais simples para projetar e implementar, mas não produzem tão bom desempenho.

Pipe em implementações tradicionais, um pipe é um stream de dados unidirecional, first-in first-out ("o primeiro a chegar é o primeiro a sair"), não-estruturado e de tamanho máximo definido. Processos escritores adicionam dados em um lado do pipe; processos leitores recuperam dados do outro lado do pipe. Uma vez lido, o dado é removido do pipe, ou seja, não fica disponível para outros leitores. Pipes provêm um mecanismo simples de controle de fluxo. Um processo tentando ler um pipe vazio é bloqueado até que mais dados sejam escritos para o pipe. Do mesmo modo, um processo tentando escrever para um pipe cheio é bloqueado até que um outro processo leia (e conseqüentemente remova) dados do pipe.

Buffer cache Uma *cache* de sistema de arquivos normalmente é dividida em *buffers*, daí o nome *buffer cache*, dado a este tipo de *cache*. O objetivo principal do buffer cache é aumentar o desempenho dos sistemas de arquivos, eliminando diversos acessos ao disco. Além disso, no Linux, esta cache conta com algoritmos de leitura antecipada ("read ahead") e escalonamento de gravações de blocos.

Spin locks a primitiva de bloqueio (ou trava) mais simples é um *spin lock* também chamado uma trava simples ("simple lock") ou um mutex simples("simple

mutex"). Se um recurso está protegido por um spin lock, uma thread tentando adquirir o recurso irá entrar em espera-ocupada ("busy-wait") através de um loop, até que o recurso seja desbloqueado. Um spin lock é usualmente uma variável escalar que é zero se disponível e um se bloqueada. A variável é manipiulada usando um loop de espera-ocupada através de uma instrução test-and-set ou similar disponível na máquina.

NFS

A Sun Microsystems introduziu o NFS (Network File Systems - "Sistema de Arquivo de Redes") em 1985 como um meio de prover acesso transparente a sistemas de arquivos remotos. Além de publicar o protocolo, a Sun também licenciou uma referência de implementação, a qual foi usada por vendedores para portar NFS para vários sistemas operacionais. NFS desde então tornou-se o padrão de fato da indústria, suportado virtualmente por toda variante UNIX e vários sistemas não-UNIX.

A arquitetura NFS é baseada em um modelo *cliente-servidor*. Um servidor de arquivos é uma máquina que exporta um conjunto de arquivos. Clientes são máquinas que acessam tais arquivos. Uma única máquina pode atuar tanto como um servidor quanto como um cliente para sistemas de arquivos diferentes. O código NFS, entretanto, é dividido em porções cliente e servidor, permitindo sistemas somente cliente ou somente servidor.

Clientes e servidores comunicam-se através de chamadas de procedimentos remoto (RPC), a qual opera como requisições síncronas. Quando uma aplicação no cliente tenta acessar um arquivo remoto, o kernel envia uma requisição para o servidor, e o processo cliente bloqueia até que ele receba uma resposta. O servidor espera por requisições de clientes, as processa, e envia respostas de volta para os clientes.

A característica mais importante do protocolo NFS é que o servidor é stateless.

RFS

A AT&T introduziu o sistema de arquivos RFS (Remote File Sharing - "Arquivos Compartilhados Remotamente") no UNIX SVR3 para prover acesso a arquivos remotos através de uma rede. Seu objetivo básico é similar ao do NFS, mas RFS tem uma arquitetura e projeto fundamentalmente diferentes.

Similar a NFS, RFS é baseado no modelo cliente-servidor. O servidor exporta diretórios, e o cliente os monta. Qualquer máquina pode ser um cliente ou um servidor, ou ambos. As similaridades terminam aqui. RFS é uma

arquitetura completamente stateful, o que é necessário para prover corretamente semânticas de abertura de arquivos UNIX. Isto tem um grande impacto em sua implementação e em sua funcionalidade.

Procedimentos stubs Os procedimentos *stubs* são talvez uma das partes mais importantes de um sistema de RPC. Sua função é isolar o programador dos detalhes referentes à comunicação através da rede.

Operação de uma RPC O mecanismo de RPC tem que ser transparente, isto é, para o programador as chamadas têm que parecer locais. Para tanto, é necessário introduzir alguns elementos que vão esconder as complexidades inerentes ao problema.

Ao ser feita uma chamada a um procedimento remoto, da mesma forma que em uma chamada local, os parâmetros são colocados na pilha, asim como é guardado o endereço de retorno. Entretanto, ao invés de se chamar o procedimento propriamente dito, é invocado um *client stub*, o qual coloca os parâmetros em uma mensagem *marshalling* juntamente com uma identificação do procedimento chamado, e envia a mensagem ao servidor, bloqueando-se, em seguida, até que chegue a resposta. Quando esta chega ao lado servidor, o *dispatcher* ("despachante") identifica qual o procedimento sendo solicitado e passa a tarefa ao server *stub* apropriado, o qual retira os parâmetros da mensagem ("unmarshalling") e chama o procedimento servidor "real" da maneira usual, ou seja, colocando os parâmetros na pilha e passando o controle ao mesmo.

O procedimento servidor, então, executa o trabalho que lhe compete e retorna os resultados ao *server stub*, o qual compõe uma nova mensagem e a envia de volta ao cliente. O *cliente stub* irá recebê-la e fazer um retorno da maneira usual à rotina que invocou a RPC.

Largura de banda O desempenho de uma rede é medido de duas maneiras fundamentais: *bandwidth* ("largura de banda", também chamado *throughput* - *"taxa de transferência de dados"*) e *latência* (também chamado *delay* - *"atraso"*). O bandwidth de uma rede é dado pelo número de bits que podem ser transmitidos através da rede em uma certo período de tempo. Por exemplo, uma rede pode ter um bandwidth de 10 milhões de bits/segundo (Mbps), significando que ela é capaz de entregar 10 milhões de bits a cada segundo. É algumas vezes útil pensarmos sobre bandwidth em termos de quanto tempo ele leva para transmitir cada bit de dados. Em uma rede de 10Mbps, por exemplo, ele leva 0.1 microssegundos (μs) para transmitir cada bit.

Bandwidth e throughput são os dois termos mais confusos usados em redes. Tende-se a usar a palavra *throughput* como referência a *medida de desempenho* de um sistema. Conseqüentemente, por causa das várias ineficiências de implementação, um par de nós conectados por um link com um bandwidth de 10Mbps pode obter um throughput de somente 2Mbps. Isto significa que uma aplicação em um host pode enviar dados a outro host em 2Mbps.

No trabalho usamos o termo "taxa" seguido da unidade de medida (por exemplo, MB/s) como referência a bandwidth.

Latência

A segunda métrica de desempenho - latência - corresponde a quanto tempo leva um único bit para propagar de um lado de uma rede para outro. Latência é medida estritamente em termos de tempo. Por exemplo, uma rede transcontinental pode ter uma latência de 24 milisegundos (ms), isto é, um único bit leva 24ms para viajar de um lado do país ao outro, Há muitas situações nas quais é mais importante saber quanto tempo leva para enviar um bit de um lado a outro de uma rede e a volta, ao contrário do que uma latência uni- direcional. Chama-se isto de *round-trip time* ("tempo de ida e volta") (RTT) da rede.

A.3 O sistema de arquivos UNIX

O sistema de arquivos UNIX é importante porque ele tem várias características que o tornam mais fácil de usar e compartilhar informações, e é largamente usado. Esta seção revisa as principais características do sistema de arquivos UNIX que se relacionam ao resto da dissertação.

A.3.1 Sistema de nomes hierárquico

O sistema de arquivos UNIX é organizado como uma árvore hierárquica de diretórios e arquivos. A árvore começa em um diretório distinto chamado raiz (root). Um sistema de nomes hierárquico é criado permitindo entradas em um diretório tanto de nomes de arquivos quanto de outros diretórios. Esta definição recursiva do espaço de nomes originou-se em Multics e é usado no UNIX e na maior parte dos sistemas operacionais modernos.

Objetos têm um *pathname* que é uma seqüência de *componentes* separadas pelo caractere "/", por exemplo, "a/b/c". O diretório raiz é chamado "/". *Resolução de nomes* é o processo de examinar a estrutura do diretório para seguir o endereço do destinatário. Este

é um procedimento interativo onde o diretório corrente é lido para obter o próximo componente do pathname. Se o componente é encontrado e há mais componentes deixados para processar, então o diretório atual é avançado e a busca continua. A busca termina com sucesso quando o último componente é encontrado. Por exemplo, o pathname "a/b/c" define um caminho através da hierarquia que começa no diretório raíz, "/", e procede ao diretório "a", e segue para o diretório "b", e termina no objeto "c". "a/b/c" pode ser o pathname de um arquivo, diretório, pseudo-dispositivo, ou uma ligação simbólica para um outro pathname. (links simbólicos são explicados abaixo.)

Como uma comodidade para usuários, UNIX provê um *diretório de trabalho corrente* e pathnames *relativos*. Pathnames relativos implicitamente começam no diretório de trabalho corrente, enquanto pathnames *absolutos* começam no diretório raiz. Pathnames relativos são distintos porque eles não começam com "/", o nome do diretório raiz. Por exemplo, se o diretório corrente é "a/b", então o objeto "a/b/c" pode ser referenciado pelo pathname relativo "c". O diretório de trabalho corrente pode ser alterado pelo processo para tornar mais fácil nomear arquivos comumente usados. Nomes relativos são curtos, o que significa que eles são mais fáceis de digitar pelo usuário e mais eficientes para processar pelo sistema operacional.

A combinação de pathnames relativos e a especificação do diretório pai significa que um pathname pode começar em qualquer ponto na hierarquia e viajar para qualquer outro ponto. O pai de um diretório pode ser referenciado com o nome "..". Por exemplo, se o diretório de trabalho corrente é "a/b/c", então ".." faz referência a "a/b". Diretórios irmãos são facilmente referenciados com nomes relativos (por exemplo, "../sibling"). Consequentemente, pathnames relativos podem ser usados dentro de um conjunto de diretórios relacionados de maneira que a localização absoluta dos diretórios não é importante, somente, sua localização relativa.

A estrutura hierárquica do espaço de nomes pode tornar-se mais arbitrariamente conectada usando *links simbólicos*. Um link simbólico é um arquivo especial que é uma referência pelo nome a outro ponto na hierarquia. Não há restrições no alvo de um link simbólico, de maneira que hierarquias planas podem ser convertidas a um grafo dirigido geral. Links simbólicos são implementados como arquivos que contém pathnames. Atravessar um link é implementado substituindo o nome do link com seu conteúdo e continuando o algoritmo de busca normal.

Bibliografia

[Accetta86] M .Accetta et al. Mach: A New Kernel Foundation for UNIX Development. *Proc. Summer 1986 USENIX Conf.*, pages 93–112, 1986.

[Baker96] Kevin Lai and Mary Baker. A Performance Comparison of UNIX Operating Systems on the Pentium. *Proceedings of the 1996 USENIX Technical Conference*, page 1, January 1996.

[Cornes97] Phil Cornes. *Linux A-Z.* Prentice-Hall, 1997.

[Howard88] J. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, february 1988.

[Killian84] T.J. Killian. Processes as Files. *USENIX Summer Conference Proceedings*, june 1984.

[Lampson91] Butler Lampson et al. Authentication in Distributed Systems: Theory and Practice. *Proc. 13th Symp. on Op. Sys. Princ.*, pages 165–182, 1991.

[McVoy96] Larry McVoy and Carl Staelin. Lmbench:Portable Tools for Performance Analysis. *Proceedings of the 1996 USENIX Technical Conference*, pages 279–295, January 1996.

[Miller87] S.P. Miller et al. Kerberos Authentication and Authorization System. *Massachusetts Institute of Technology*, 1997.

[NBS77] National Bureau of Standards (U.S.). Federal Information Processing Standard 46. *National Technical Information Service*, 1977.

[Needham82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, London, 1982.

[Ousterhout90] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *USENIX Summer Conference*, pages 247–256, 1990.

[Pike87] Rob Pike. The Text Editor *sam*. *Software - Practice and Experience*, 17:813–845, november 1987.

[Pike91] Rob Pike. 81/2, The Plan 9 Window System. *USENIX Summer Conference Proceedings*, pages 257–265, june 1991.

[Pike94] Rob Pike. Acme: A User Interface for Programmers. *USENIX Winter Conference Proceedings*, 1994.

[Pike95] Rob Pike. How to Use the Plan 9 C Compiler. *Plan 9 Programmers Manual*, 2, 1995.

[Presotto93] Dave Presotto et al. The Use of Names Spaces in Plan 9. *Op. Sys. Rev.*, 27(2):72–76, april 93.

[Sandberg85] R. Sandberg et al. Design and Implementation of the SUN Network File System. *USENIX Conference Proceedings*, pages 119–130, june 1985.

[Seltzer97] Aaron B. Brown and Margo I. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. *Sigmetrics*, 1997.

[Tanenbaum90] A.S. Tanenbaum et al. Experiences with the Amoeba Distributed Operating Systems. *Comms. ACM*, 33(12):46–63, 1990.

[Trickey95] Howard Trickey. APE - The ANSI/POSIX Environment. *Plan 9 Programmers Manual*, 2, 1995.

[Winterbottom93] Dave Pressoto and Phil Winterbottom. The Organization of Networks in Plan 9. *USENIX Proc. of the Winter 1993 Conf.*, pages 43–50, 1993.

[Winterbottom94] Phil Winterbottom. Acid: A Debugger Built from a Language. *USENIX Winter Proceedings*, 1994.